

May 1980

An Introduction to the Intel®  
MCS-51™ Single-Chip  
Microcomputer Family

John Wharton  
Microcontroller Applications

---

# AN INTRODUCTION TO THE INTEL MCS-51™ SINGLE-CHIP MICROCOMPUTER FAMILY

## Contents

<b>1. INTRODUCTION</b>	1
Family Overview	1
Microcomputer Background Concepts	3
<b>2. ARCHITECTURE AND ORGANIZATION</b>	4
Central Processing Unit	5
Memory Spaces	8
Input/Output Ports	9
Special Peripheral Functions	9
<b>3. INSTRUCTION SET AND ADDRESSING     MODES</b>	14
Data Addressing Modes	14
Addressing Mode Combinations	17
Advantages of Symbolic Addressing	17
Arithmetic Instruction Usage	18
Multiplication and Division	19
Logical Byte Operations	19
Program Control	20
Operate-and-Branch Instructions	21
Stack Operations	21
Table Look-Up Instructions	22
<b>4. BOOLEAN PROCESSING INSTRU-     TIONS</b>	24
Direct Bit Addressing	24
Bit Manipulation Instructions	24
Combinatorial Logic Equations	25
<b>5. ON-CHIP PERIPHERAL FUNCTIONS</b>	27
I/O Ports	27
Serial Port and Timer	28
<b>6. SUMMARY</b>	29

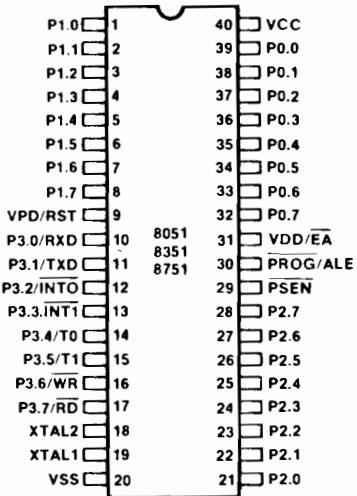


Figure 1a. 8051 Microcomputer Pinout Diagram

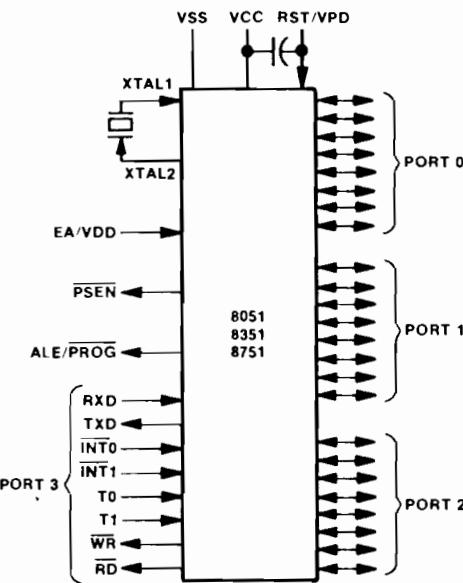


Figure 1b. 8051 Microcomputer Logic Symbol

## 1. INTRODUCTION

In 1976 Intel introduced the MCS-48™ family, consisting of the 8048, 8748, and 8035 microcomputers. These parts marked the first time a complete microcomputer system, including an eight-bit CPU, 1024 8-bit words of ROM or EPROM program memory, 64 words of data memory, I/O ports and an eight-bit timer/counter could be integrated onto a single silicon chip. Depending only on the program memory contents, one chip could control a limitless variety of products, ranging from appliances or automobile engines to text or data processing equipment. Follow-on products stretched the MCS-48™ architecture in several directions: the 8049 and 8039 doubled the amount of on-chip memory and ran 83% faster; the 8021 reduced costs by executing a subset of the 8048 instructions with a somewhat slower clock; and the 8022 put a unique two-channel 8-bit analog-to-digital converter on the same NMOS chip as the computer, letting the chip interface directly with analog transducers.

Now three new high-performance single-chip microcomputers—the Intel® 8051, 8751, and 8031—extend the advantages of Integrated Electronics to whole new product areas. Thanks to Intel's new HMOS technology, the MCS-51™ family provides four times the program memory and twice the data memory as the 8048 on a single chip. New I/O and peripheral capabilities both increase the range of applicability and reduce total system cost. Depending on the use, processing throughput increases by two and one-half to ten times.

This Application Note is intended to introduce the reader to the MCS-51™ architecture and features. While it does not assume intimacy with the MCS-48™ product line on the part of the reader, he/she should be familiar with

some microprocessor (preferably Intel's, of course) or have a background in computer programming and digital logic.

## Family Overview

Pinout diagrams for the 8051, 8751, and 8031 are shown in Figure 1. The devices include the following features:

- Single-supply 5 volt operation using HMOS technology.
- 4096 bytes program memory on-chip (not on 8031).
- 128 bytes data memory on-chip.
- Four register banks.
- 128 User-defined software flags.
- 64 Kilobytes each program and external RAM addressability.
- One microsecond instruction cycle with 12 MHz crystal.
- 32 bidirectional I/O lines organized as four 8-bit ports (16 lines on 8031).
- Multiple mode, high-speed programmable Serial Port.
- Two multiple mode, 16-bit Timer/Counters.
- Two-level prioritized interrupt structure.
- Full depth stack for subroutine return linkage and data storage.
- Augmented MCS-48™ instruction set.
- Direct Byte and Bit addressability.
- Binary or Decimal arithmetic.
- Signed-overflow detection and parity computation.
- Hardware Multiple and Divide in 4 usec.
- Integrated Boolean Processor for control applications.
- Upwardly compatible with existing 8048 software.

All three devices come in a standard 40-pin Dual In-Line Package, with the same pin-out, the same timing, and the same electrical characteristics. The primary difference between the three is the on-chip program memory: different types are offered to satisfy differing user requirements.

The 8751 provides 4K bytes of ultraviolet-Erasable, Programmable Read Only Memory (EPROM) for program development, prototyping, and limited production runs. (By convention, 1K means  $2^{10} = 1024$ . 1k with a lower case "k"—equals  $10^3 = 1000$ .) This part may be individually programmed for a specific application using Intel's Universal PROM Programmer (UPP). If software bugs are detected or design specifications change the same part may be "erased" in a matter of minutes by exposure to ultraviolet light and reprogrammed with the modified code. This cycle may be repeated indefinitely during the design and development phase.

The final version of the software must be programmed into a large number of production parts. The 8051 has 4K bytes of ROM which are mask-programmed with the customer's order when the chip is built. This part is considerably less expensive, but cannot be erased or altered after fabrication.

The 8031 does not have any program memory on-chip, but may be used with up to 64K bytes of external standard or multiplexed ROMs, PROMs, or EPROMs. The 8031 fits well in applications requiring significantly larger or smaller amounts of memory than the 4K bytes provided by its two siblings.

(The 8051 and 8751 automatically access external program memory for all addresses greater than the 4096 bytes on-chip. The External Access input is an override for all internal program memory—the 8051 and 8751 will each emulate an 8031 when pin 31 is low.)

Throughout this Note, "8051" is used as a generic term. Unless specifically stated otherwise, the point applies equally to all three components. Table 1 summarizes the quantitative differences between the members of the MCS-48™ and MCS-51™ families.

The remainder of this Note discusses the various MCS-51™ features and how they can be used. Software and/or hard-

ware application examples illustrate many of the concepts. Several isolated tasks (rather than one complete system design example) are presented in the hope that some of them will apply to the reader's experiences or needs.

A document this short cannot detail all of a computer system's capabilities. By no means will all the 8051 instructions be demonstrated; the intent is to stress new or unique MCS-51™ operations and instructions generally used in conjunction with each other. For additional hardware information refer to the Intel **MCS-51™ Family User's Manual**, publication number 121517. The assembly language and use of ASM51, the MCS-51™ assembler, are further described in the **MCS-51™ Macro Assembler User's Guide**, publication number 9800937.

The next section reviews some of the basic concepts of microcomputer design and use. Readers familiar with the 8048 may wish to skim through this section or skip directly to the next, "ARCHITECTURE AND ORGANIZATION."

#### Microcomputer Background Concepts

Most digital computers use the binary (base 2) number system internally. All variables, constants, alphanumeric characters, program statements, etc., are represented by groups of binary digits ("bits"), each of which has the value 0 or 1. Computers are classified by how many bits they can move or process at a time.

The MCS-51™ microcomputers contain an eight-bit central processing unit (CPU). Most operations process variables eight bits wide. All internal RAM and ROM, and virtually all other registers are also eight bits wide. An eight-bit ("byte") variable (shown in Figure 2) may assume one of  $2^8 = 256$  distinct values, which usually represent integers between 0 and 255. Other types of numbers, instructions, and so forth are represented by one or more bytes using certain conventions.

For example, to represent positive and negative values, the most significant bit (D7) indicates the sign of the other seven bits—0 if positive, 1 if negative—allowing integer variables between -128 and +127. For integers with extremely large magnitudes, several bytes are manipulated together as "multiple precision" signed or unsigned integers—16, 24, or more bits wide.

Table 1. Features of Intel's Single-Chip Microcomputers

EPROM Program Memory	ROM Program Memory	External Program Memory	Program Memory (Int/Max)	Data Memory (Bytes)	Instr. Cycle Time	Input/Output Pins	Interrupt Sources	Reg. Banks
8748	8021	—	1K/1K	64	8.4 $\mu$ Sec	21	0	1
	8022	—	2K/2K	64	8.4 $\mu$ Sec	28	2	1
	8048	8035	1K/4K	64	2.5 $\mu$ Sec	27	2	2
	8049	8039	2K/4K	128	1.36 $\mu$ Sec	27	2	2
	8751	8051	4K/64K	128	1.0 $\mu$ Sec	32	5	4

The letters "MCS" have traditionally indicated a system or family of compatible Intel® microcomputer components, including CPUs, memories, clock generators, I/O expanders, and so forth. The numerical suffix indicates the microprocessor or microcomputer which serves as the cornerstone of the family. Microcomputers in the MCS-48™ family currently include the 8048-series (8035, 8048, & 8748), the 8049-series (8039 & 8049), and the 8021 and 8022; the family also includes the 8243, an I/O expander compatible with each of the microcomputers. Each computer's CPU is derived from the 8048, with essentially the same architecture, addressing modes, and instruction set, and a single assembler (ASM48) serves each.

The first members of the MCS-51™ family are the 8051, 8751, and 8031. The architecture of the 8051-series, while derived from the 8048, is not strictly compatible; there are more addressing modes, more instructions, larger address spaces, and a few other hardware differences. In this Application Note the letters "MCS-51" are used when referring to *architectural* features of the 8051-series—features which would be included on possible future microcomputers based on the 8051 CPU. Such products could have different amounts of memory (as in the 8048/8049) or different peripheral functions (as in the 8021 and 8022) while leaving the CPU and instruction set intact. ASM51 is the assembler used by all microcomputers in the 8051 family.

Two digit decimal numbers may be "packed" in an eight-bit value, using four bits for the binary code of each digit. This is called Binary-Coded Decimal (BCD) representation, and is often used internally in programs which interact heavily with human beings.

Alphanumeric characters (letters, numbers, punctuation marks, etc.) are often represented using the American Standard Code for Information Interchange (ASCII) convention. Each character is associated with a unique seven-bit binary number. Thus one byte may represent

0	1	0	1	0	0	0	1
D7	D6	D5	D4	D3	D2	D1	D0

**Figure 2. Representation of Bits Within an Eight-Bit "Byte" (Value shown = 01010001 Binary = 81 decimal).**

a single character, and a word or sequence of letters may be represented by a series (or "string") of bytes. Since the ASCII code only uses 128 characters, the most significant bit of the byte is not needed to distinguish between characters. Often D7 is set to 0 for all characters. In some coding schemes, D7 is used to indicate the "parity" of the other seven bits—set or cleared as necessary to ensure that the total number of "1" bits in the eight-bit code is even ("even parity") or odd ("odd parity"). The 8051 includes hardware to compute parity when it is needed.

A computer program consists of an ordered sequence of specific, simple steps to be executed by the CPU one-at-a-time. The method or sequence of steps used collectively to solve the user's application is called an "algorithm."

The program is stored inside the computer as a sequence of binary numbers, where each number corresponds to one of the basic operations ("opcodes") which the CPU is capable of executing. In the 8051, each program memory location is one byte. A complete instruction consists of a sequence of one or more bytes, where the first defines the operation to be executed and additional bytes (if needed) hold additional information, such as data values or variable addresses. No instruction is longer than three bytes.

The way in which binary opcodes and modifier bytes are assigned to the CPU's operations is called the computer's "machine language." Writing a program directly in machine language is time-consuming and tedious. Human beings think in words and concepts rather than encoded numbers, so each CPU operation and resource is given a name and standard abbreviation ("mnemonic"). Programs are more easily discussed using these standard mnemonics, or "assembly language," and may be typed into an Intel® Intellec® 800 or Series II® microcomputer development system in this form. The development system can mechanically translate the program from assembly language "source" form to machine language "object" code using a program called an "assembler." The MCS-51™ assembler is called ASM51.

There are several important differences between a computer's machine language and the assembly language used as a tool to represent it. The machine language or instruction set is the set of operations which the CPU can perform while a program is executing ("at run-time"), and is strictly determined by the microcomputer hardware design.

The assembly language is a standard (though more-or-less arbitrary) set of symbols including the instruction set mnemonics, but with additional features which further simplify the program design process. For example, ASM51 has controls for creating and formatting a program listing, and a number of directives for allocating variable storage and inserting arbitrary bytes of data into the object code for creating tables of constants.

In addition, ASM51 can perform sophisticated mathematical operations, computing addresses or evaluating arithmetic expressions to relieve the programmer from this drudgery. However, these calculations can only use information known at "assembly time."

For example, the 8051 performs arithmetic calculations at run-time, eight bits at a time. ASM51 can do similar operations 16 bits at a time. The 8051 can only do one simple step per instruction, while ASM51 can perform complex calculations in each line of source code. However, the operations performed by the assembler may only use parameter values fixed at assembly-time, not variables whose values are unknown until program execution begins.

For example, when the assembly language source line,

ADD A, #(LOOP\_COUNT + 1) \* 3

is *assembled*, ASM51 will find the value of the previously-defined constant "LOOP\_COUNT" in an internal symbol table, increment the value, multiply the sum by three, and (assuming it is between -256 and 255 inclusive) truncate the product to eight bits. When this instruction is *executed*, the 8051 ALU will just add that resulting constant to the accumulator.

Some similar differences exist to distinguish number system ("radix") specifications. The 8051 does all computations in binary (though there are provisions for then converting the result to decimal form). In the course of writing a program, though, it may be more convenient to specify constants using some other radix, such as base 10. On other occasions, it is desirable to specify the ASCII code for some character or string of characters without referring to tables. ASM51 allows several representations for constants, which are converted to binary as each instruction is assembled.

For example, binary numbers are represented in the

assembly language by a series of ones and zeros (naturally), followed by the letter "B" (for Binary); octal numbers as a series of octal digits (0-7) followed by the letter "O" (for Octal) or "Q" (which doesn't stand for anything, but *looks* sort of like an "O" and is less likely to be confused with a zero).

Hexadecimal numbers are represented by a series of hexadecimal digits (0-9,A-F), followed by (you guessed it) the letter "H." A "hex" number must begin with a decimal digit; otherwise it would look like a user-defined symbol (to be discussed later). A "dummy" leading zero may be inserted before the first digit to meet this constraint. The character string "BACH" could be a legal label for a Baroque music synthesis routine; the string "0BACH" is the hexadecimal constant  $BAC_{16}$ . This is a case where adding 0 makes a big difference.

Decimal numbers are represented by a sequence of decimal digits, optionally followed by a "D." If a number has no suffix, it is assumed to be decimal—so it had better not contain any non-decimal digits. "0BAC" is not a legal representation for anything.

When an ASCII code is needed in a program, enclose the desired character between two apostrophes (as in '#) and the assembler will convert it to the appropriate code (in this case 23H). A string of characters between apostrophes is translated into a series of constants; 'BACH' becomes 42H, 41H, 43H, 48H.

These same conventions are used throughout the associated Intel documentation. Table 2 illustrates some of the different number formats.

## 2. ARCHITECTURE AND ORGANIZATION

Figure 3 blocks out the MCS-51™ internal organization. Each microcomputer combines a Central Processing Unit, two kinds of memory (data RAM plus program ROM or EPROM), Input/Output ports, and the mode,

Table 2. Notations Used to Represent Numbers

Bit Pattern	Binary	Octal	Hexa-Decimal	Decimal	Signed Decimal
0 0 0 0 0 0 0 0	0B	0Q	00H	0	0
0 0 0 0 0 0 0 1	1B	1Q	01H	1	+1
.....	..	..	..	..	...
0 0 0 0 0 1 1 1	111B	7Q	07H	7	+7
0 0 0 0 1 0 0 0	1000B	10Q	08H	8	+8
0 0 0 0 1 0 0 1	1001B	11Q	09H	9	+9
0 0 0 0 1 0 1 0	1010B	12Q	0AH	10	+10
.....	..	..	..	..	...
0 0 0 0 1 1 1 1	1111B	17Q	0FH	15	+15
0 0 0 1 0 0 0 0	10000B	20Q	10H	16	+16
.....	..	..	..	..	...
0 1 1 1 1 1 1 1	111111B	177Q	7FH	127	+127
1 0 0 0 0 0 0 0	1000000B	200Q	80H	128	-128
1 0 0 0 0 0 0 1	1000001B	201Q	81H	129	-127
.....	..	..	..	..	...
1 1 1 1 1 1 1 0	1111110B	376Q	0FEH	254	-2
1 1 1 1 1 1 1 1	1111111B	377Q	0FFH	255	-1

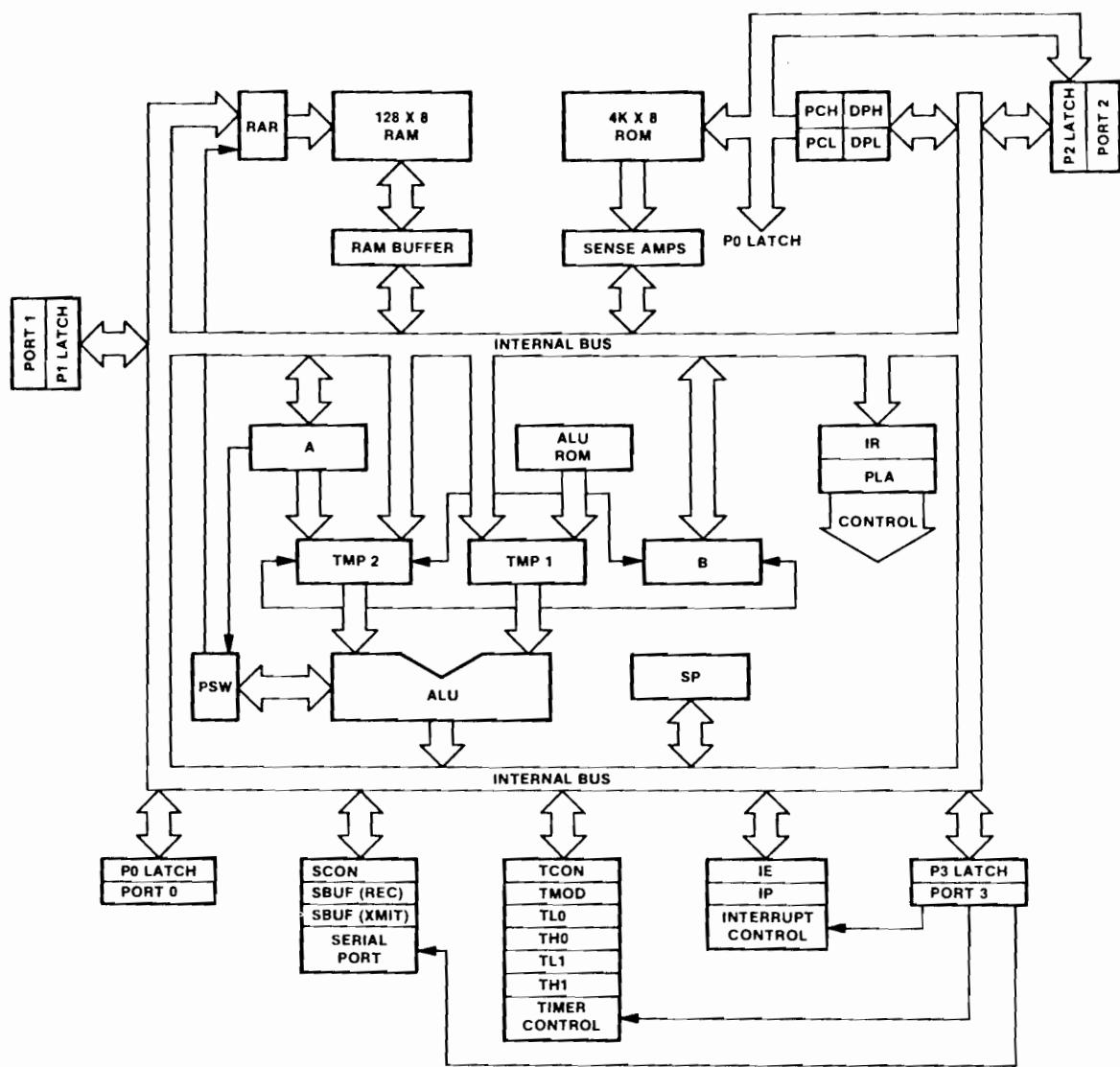


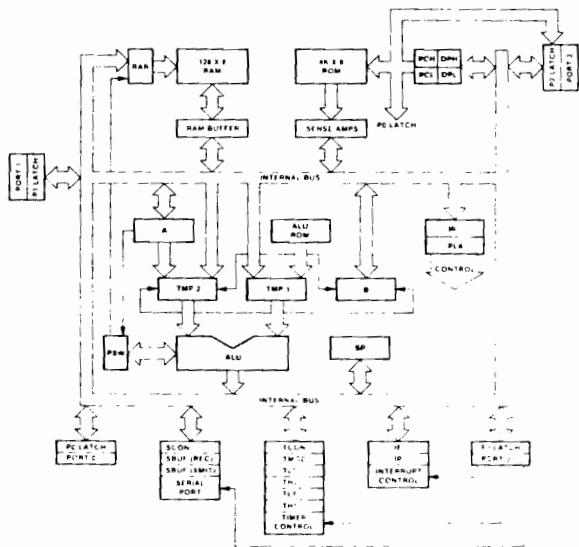
Figure 3. Block Diagram of 8051 Internal Structure

status, and data registers and random logic needed for a variety of peripheral functions. These elements communicate through an eight-bit data bus which runs throughout the chip, somewhat akin to indoor plumbing. This bus is buffered to the outside world through an I/O port when memory or I/O expansion is desired.

Let's summarize what each block does; later chapters dig into the CPU's instruction set and the peripheral registers in much greater detail.

### Central Processing Unit

The CPU is the “brains” of the microcomputer, reading the user's program and executing the instructions stored therein. Its primary elements are an eight-bit Arithmetic/Logic Unit with associated registers A, B, PSW, and SP, and the sixteen-bit Program Counter and “Data Pointer” registers.



### Arithmetic Logic Unit

The ALU can perform (as the name implies) arithmetic and logic functions on eight-bit variables. The former include basic addition, subtraction, multiplication, and division; the latter include the logical operations AND, OR, and Exclusive-OR, as well as rotate, clear, complement, and so forth. The ALU also makes conditional branching decisions, and provides data paths and temporary registers used for data transfers within the system. Other instructions are built up from these primitive functions: the addition capability can increment registers or automatically compute program destination addresses; subtraction is also used in decrementing or comparing the magnitude of two variables.

These primitive operations are automatically cascaded and combined with dedicated logic to build complex instructions such as incrementing a sixteen-bit register pair. To execute one form of the compare instruction, for example, the 8051 increments the program counter three times, reads three bytes of program memory, computes a register address with logical operations, reads internal data memory twice, makes an arithmetic comparison of two variables, computes a sixteen-bit destination address, and decides whether or not to make a branch—all in two microseconds!

An important and unique feature of the MCS-51 architecture is that the ALU can also manipulate one-bit as well as eight-bit data types. Individual bits may be set, cleared, or complemented, moved, tested, and used in logic computations. While support for a more primitive data type may initially seem a step backwards in an era of increasing word length, it makes the 8051 especially well suited for controller-type applications. Such algorithms *inherently* involve Boolean (true/false) input and output variables, which were heretofore difficult to implement with standard microprocessors. These features are collectively referred to as the MCS-51™ “Boolean Processor,” and are described in the so-named chapter to come.

Thanks to this powerful ALU, the 8051 instruction set fares well at both real-time control and data intensive algorithms. A total of 51 separate operations move and manipulate three data types: Boolean (1-bit), byte (8-bit), and address (16-bit). All told, there are eleven addressing modes—seven for data, four for program sequence control (though only eight are used by more than just a few specialized instructions). Most operations allow several addressing modes, bringing the total number of instructions (operation/addressing mode combinations) to 111, encompassing 255 of the 256 possible eight-bit instruction opcodes.

### Instruction Set Overview

Table 4 lists these 111 instructions classified into five groups:

- Arithmetic Operations
- Logical Operations for Byte Variables
- Data Transfer Instructions
- Boolean Variable Manipulation
- Program Branching and Machine Control

MCS-48™ programmers perusing Table 4 will notice the absence of special categories for Input/Output, Timer/Counter, or Control instructions. These functions are all still provided (and indeed many new functions are added), but as special cases of more generalized operations in other categories. To explicitly list all the useful instructions involving I/O and peripheral registers would require a table approximately four times as long.

Observant readers will also notice that all of the 8048's page-oriented instructions (conditional jumps, JMPP, MOVP, MOVP3) have been replaced with corresponding but non-paged instructions. The 8051 instruction set is entirely *non*-page-oriented. The MCS-48™ “MOVP” instruction replacement and all conditional jump instructions operate relative to the program counter, with the actual jump address computed by the CPU during instruction execution. The “MOVP3” and “JMPP” replacements are now made relative to another sixteen-bit register, which allows the effective destination to be anywhere in the program memory space, regardless of where the instruction itself is located. There are even three-byte jump and call instructions allowing the destination to be *anywhere* in the 64K program address space.

The instruction set is designed to make programs efficient both in terms of code size and execution speed. No instruction requires more than three bytes of program memory, with the majority requiring only one or two bytes. Virtually all instructions execute in either one or two instruction cycles—one or two microseconds with a 12-MHz crystal—with the sole exceptions (multiply and divide) completing in four cycles.

Many instructions such as arithmetic and logical functions or program control, provide both a short and a long form for the same operation, allowing the programmer to optimize the code produced for a specific application. The 8051 usually fetches two instruction bytes per instruction cycle, so using a shorter form can lead to faster execution as well.

For example, any byte of RAM may be loaded with a constant with a three-byte, two-cycle instruction, but the commonly used “working registers” in RAM may be initialized in one cycle with a two-byte form. Any bit anywhere on the chip may be set, cleared, or complemented by a single three-byte logical instruction using two cycles. But critical control bits, I/O pins, and software flags may be controlled by two-byte, single cycle instructions. While three-byte jumps and calls can “go anywhere” in program memory, nearby sections of code may be reached by shorter relative or absolute versions.

(MSB) (LSB)													
CY	AC	F0	RS1	RS0	OV	—	P						
<b>Symbol</b>	<b>Position</b>	<b>Name and Significance</b>											
CY	PSW.7	Carry flag. Set/cleared by hardware or software during certain arithmetic and logical instructions.											
AC	PSW.6	Auxiliary Carry flag. Set/cleared by hardware during addition or subtraction instructions to indicate carry or borrow out of bit 3.											
F0	PSW.5	Flag 0 Set/cleared/tested by software as a user-defined status flag.											
RS1	PSW.4	Register bank Select control bits 1 & 0. Set/cleared by software to determine working register bank (see Note).											
RS	PSW.3												
<b>Note—</b> the contents of (RS1, RS0) enable the working register banks as follows:													
(0,0) — Bank 0 (00H-07H) (0,1) — Bank 1 (08H-0FH) (1,0) — Bank 2 (10H-17H) (1,1) — Bank 3 (18H-1FH)													

Figure 4. PSW—Program Status Word Organization

A significant side benefit of an instruction set more powerful than those of previous single-chip microcomputers is that it is easier to generate applications-oriented software. Generalized addressing modes for byte and bit instructions reduce the number of source code lines written and debugged for a given application. This leads in turn to proportionately lower software costs, greater reliability, and faster design cycles.

#### Accumulator and PSW

The 8051, like its 8048 predecessor, is primarily an accumulator-based architecture: an eight-bit register called the accumulator ("A") holds a source operand and receives the result of the arithmetic instructions (addition, subtraction, multiplication, and division). The accumulator can be the source or destination for logical operations and a number of special data movement instructions, including table look-ups and external RAM expansion. Several functions apply exclusively to the accumulator: rotates, parity computation, testing for zero, and so on.

Many instructions implicitly or explicitly affect (or are affected by) several status flags, which are grouped together to form the Program Status Word shown in Figure 4.

(The period within entries under the Position column is called the "dot operator," and indicates a particular bit position within an eight-bit byte. "PSW.5" specifies bit 5 of the PSW. Both the documentation and ASM51 use this notation.)

The most "active" status bit is called the carry flag (abbreviated "C"). This bit makes possible multiple precision arithmetic operations including addition, subtraction,

and rotates. The carry also serves as a "Boolean accumulator" for one-bit logical operations and bit manipulation instructions. The overflow flag (OV) detects when arithmetic overflow occurs on signed integer operands, making two's complement arithmetic possible. The parity flag (P) is updated after every instruction cycle with the even-parity of the accumulator contents.

The CPU does not control the two register-bank select bits, RS1 and RS0. Rather, they are manipulated by software to enable one of the four register banks. The usage of the PSW flags is demonstrated in the Instruction Set chapter of this Note.

Even though the architecture is accumulator-based, provisions have been made to bypass the accumulator in common instruction situations. Data may be moved from any location on-chip to any register, address, or indirect address (and vice versa), any register may be loaded with a constant, etc., all without affecting the accumulator. Logical operations may be performed against registers or variables to alter fields of bits—without using or affecting the accumulator. Variables may be incremented, decremented, or tested without using the accumulator. Flags and control bits may be manipulated and tested without affecting anything else.

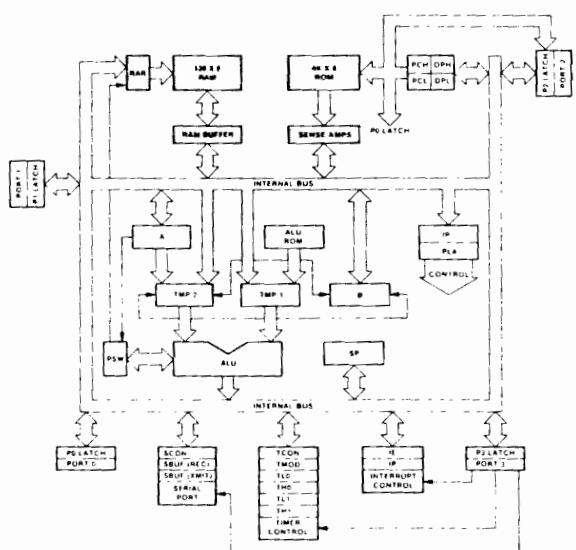
#### Other CPU Registers

A special eight-bit register ("B") serves in the execution of the multiply and divide instructions. This register is used in conjunction with the accumulator as the second input operand and to return eight-bits of the result.

The MCS-51 family processors include a hardware stack within internal RAM, useful for subroutine linkage,

passing parameters between routines, temporary variable storage, or saving status during interrupt service routines. The Stack Pointer (SP) is an eight-bit pointer register which indicates the address of the last byte pushed onto the stack. The stack pointer is automatically incremented or decremented on all push or pop instructions and all subroutine calls and returns. In theory, the stack in the 8051 may be up to a full 128 bytes deep. (In practice, even simple programs would use a handful of RAM locations for pointers, variables, and so forth—reducing the stack depth by that number.) The stack pointer defaults to 7 on reset, so that the stack will start growing up from location 8, just like in the 8048. By altering the pointer contents the stack may be relocated anywhere within internal RAM.

Finally, a 16-bit register called the data pointer (DPTR) serves as a base register in indirect jumps, table look-up instructions, and external data transfers. The high- and low-order halves of the data pointer may be manipulated as separate registers (DPH and DPL, respectively) or together using special instructions to load or increment all sixteen bits. Unlike the 8048, look-up tables can therefore start anywhere in program memory and be of arbitrary length.



## Memory Spaces

Program memory is separate and distinct from data memory. Each memory type has a different addressing mechanism, different control signals, and a different function.

The program memory array (ROM or EPROM), like an elephant, is extremely large and never forgets information, even when power is removed. Program memory is used for information needed each time power is applied: initialization values, calibration constants, keyboard layout tables, etc., as well as the program itself. The program memory has a sixteen-bit address bus; its elements

are addressed using the Program Counter or instructions which generate a sixteen-bit address.

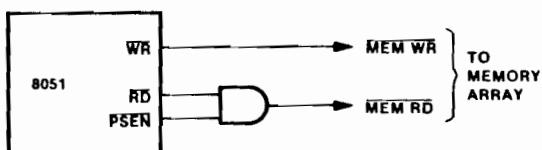
To stretch our analogy just a bit, data memory is like a mouse: it is smaller and therefore quicker than program memory, and it goes into a random state when electrical power is applied. On-chip data RAM is used for variables which are determined or may change while the program is running.

A computer spends most of its time manipulating variables, not constants, and a relatively small number of variables at that. Since eight-bits is more than sufficient to uniquely address 128 RAM locations, the on-chip RAM address register is only one byte wide. In contrast to the program memory, data memory accesses need a single eight-bit value—a constant or another variable—to specify a unique location. Since this is the basic width of the ALU and the different memory types, those resources can be used by the addressing mechanisms, contributing greatly to the computer's operating efficiency.

The partitioning of program and data memory is extended to off-chip memory expansion. Each may be added independently, and each uses the same address and data busses, but with different control signals. External program memory is gated onto the external data bus by the **PSEN** (Program Store Enable) control output, pin 29. External data memory is read onto the bus by the **RD** output, pin 17, and written with data supplied from the microcomputer by the **WR** output, pin 16. (There is no control pin to write external program ROM, which is by definition Read Only.) While both types may be expanded to up to 64K bytes, the external data memory may optionally be expanded in 256 byte “pages” to preserve the use of P2 as an I/O port. This is useful with a relatively small expansion RAM (such as the Intel® 8155) or for addressing external peripherals.

Single-chip controller programs are finalized during the project design cycle, and are not modified after production. Intel's single-chip microcomputers are not “von Neumann” architectures common among main-frame and mini-computer systems: the MCS-51™ processor *data* memory—on-chip and external—may *not* be used for program code. Just as there is no write-control signal for program memory, there is no way for the CPU to execute instructions out of RAM. In return, this concession allows an architecture optimized for efficient controller applications: a large, fixed program located in ROM, a hundred or so variables in RAM, and different methods for efficiently addressing each.

(Von Neumann machines are helpful for software development and debug. An 8051 system could be modified to have a single off-chip memory space by gating together the two memory-read controls (**PSEN** and **RD**) with a two-input AND gate (Figure 5). The CPU could then write data into the common memory array using **WR** and



**Figure 5. Combining External Program and Data Memory Arrays**

external data transfer instructions, and read instructions or data with the AND gate output and data transfer or program memory look-up instructions.)

In addition to the memory arrays, there is (yet) another (albeit sparsely populated) physical address space. Connected to the internal data bus are a score of special-purpose eight-bit registers scattered throughout the chip. Some of these—B, SP, PSW, DPH, and DPL—have been discussed above. Others—I/O ports and peripheral function registers—will be introduced in the following sections. Collectively, these registers are designated as the “special-function register” address space. Even the accumulator is assigned a spot in the special-function register address space for additional flexibility and uniformity.

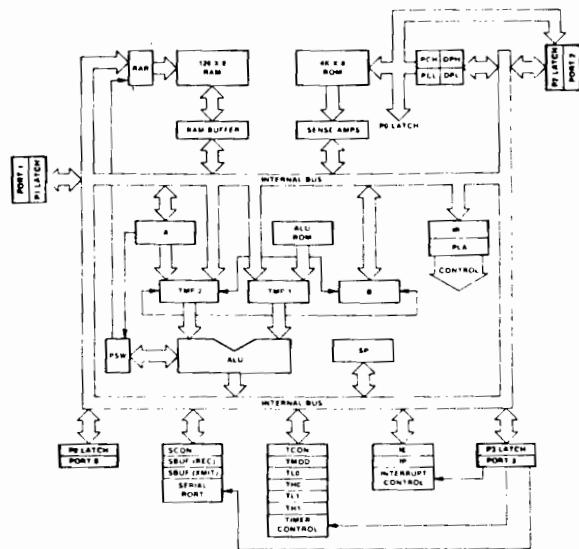
Thus, the MCS-51™ architecture supports several distinct "physical" address spaces, functionally separated at the hardware level by different addressing mechanisms, read and write control signals, or both:

- On-chip program memory;
- On-chip data memory;
- Off-chip program memory;
- Off-chip data memory;
- On-chip special-function registers.

What the *programmer sees*, though, are "logical" address spaces. For example, as far as the programmer is concerned, there is only one type of program memory, 64K bytes in length. The fact that it is formed by combining on- and off-chip arrays (split 4K/60K on the 8051 and 8751) is "invisible" to the programmer; the CPU automatically fetches each byte from the appropriate array, based on its address.

(Presumably, future microcomputers based on the MCS-51™ architecture may have a different physical split, with more or less of the 64K total implemented on-chip. Using the MCS-48™ family as a precedent, the 8048's 4K potential program address space was split 1K/3K between on- and off-chip arrays; the 8049's was split 2K/2K.)

Why go into such tedious details about address spaces? The logical addressing modes are described in the Instruction Set chapter in terms of physical address spaces. Understanding their differences now will pay off in understanding and using the chips later.



## Input/Output Ports

The MCS-51™ I/O port structure is extremely versatile. The 8051 and 8751 each have 32 I/O pins configured as four eight-bit parallel ports (P0, P1, P2, and P3). Each pin will input or output data (or both) under software control, and each may be referenced by a wide repertoire of byte and bit operations.

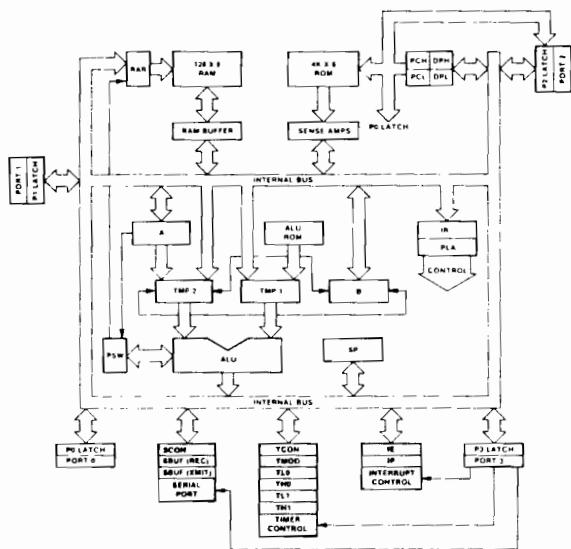
In various operating or expansion modes, some of these I/O pins are also used for special input or output functions. Instructions which access external memory use Port 0 as a multiplexed address/data bus: at the beginning of an external memory cycle eight bits of the address are output on P0; later data is transferred on the same eight pins. External data transfer instructions which supply a sixteen-bit address, and any instruction accessing external program memory, output the high-order eight bits on P2 during the access cycle. (The 8031 *always* uses the pins of P0 and P2 for external addressing, but P1 and P3 are available for standard I/O.)

The eight pins of Port 3 (P3) each have a special function. Two external interrupts, two counter inputs, two serial data lines, and two timing control strobes use pins of P3, as described in Figure 6. Port 3 pins corresponding to functions not used are available for conventional I/O.

Even within a single port, I/O functions may be combined in many ways: input and output may be performed using different pins at the same time, or the same pins at different times; in parallel in some cases, and in serial in others; as test pins, or (in the case of Port 3) as additional special functions.

(MSB) _____ (LSB)							
Symbol	Position	Name and Significance					
RD	P3.7	Read data control output. Active low pulse generated by hardware when external data memory is read.	INT1	P3.3	Interrupt 1 input pin. Low-level or falling-edge triggered.		
WR	P3.6	Write data control output. Active low pulse generated by hardware when external data memory is written.	INT0	P3.2	Interrupt 0 input pin. Low-level or falling-edge triggered.		
T1	P3.5	Timer/counter 1 external input or test pin.	TXD	P3.1	Transmit Data pin for serial port in UART mode. Clock output in shift register mode.		
T0	P3.4	Timer/counter 0 external input or test pin.	RXD	P3.0	Receive Data pin for serial port in UART mode. Data I/O pin in shift register mode.		

Figure 6. P3—Alternate Special Functions of Port 3



### Special Peripheral Functions

There are a few special needs common among control-oriented computer systems:

- keeping track of elapsed real-time;
- maintaining a count of signal transitions;
- measuring the precise width of input pulses;
- communicating with other systems or people;
- closely monitoring asynchronous external events.

Until now, microprocessor systems needed peripheral chips such as timer/counters, USARTs, or interrupt controllers to meet these needs. The 8051 integrates all of these capabilities on-chip!

### Timer/Counters

There are two sixteen-bit multiple-mode Timer/Counters on the 8051, each consisting of a "High" byte (corresponding to the 8048 "T" register) and a low byte (similar to the 8048 prescaler, with the additional flexibility of being

software-accessible). These registers are called, naturally enough, TH0, TL0, TH1, and TL1. Each pair may be independently software programmed to any of a dozen modes with a mode register designated TMOD (Figure 7), and controlled with register TCON (Figure 8).

The timer modes can be used to measure time intervals, determine pulse widths, or initiate events, with one-microsecond resolution, up to a maximum interval of 65,536 instruction cycles (over 65 milliseconds). Longer delays may easily be accumulated through software. Configured as a counter, the same hardware will accumulate external events at frequencies from D.C. to 500 KHz, with up to sixteen bits of precision.

### Serial Port Interface

Each microcomputer contains a high-speed, full-duplex, serial port which is software programmable to function in four basic modes: shift-register I/O expander, 8-bit UART, 9-bit UART, or interprocessor communications link. The UART modes will interface with standard I/O devices (e.g. CRTs, teletypewriters, or modems) at data rates from 122 baud to 31 kilobaud. Replacing the standard 12 MHz crystal with a 10.7 MHz crystal allows 110 baud. Even or odd parity (if desired) can be included with simple bit-handling software routines. Inter-processor communications in distributed systems takes place at 187 kilobaud with hardware for automatic address/data message recognition. Simple TTL or CMOS shift registers provide low-cost I/O expansion at a super-fast 1 Megabaud. The serial port operating modes are controlled by the contents of register SCON (Figure 9).

### Interrupt Capability and Control

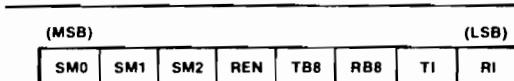
(Interrupt capability is generally considered a CPU function. It is being introduced here since, from an applications point of view, interrupts relate more closely to peripheral and system interfacing.)

	(MSB)					(LSB)		M1	M0	Operating Mode
	GATE	C/T M1 M0 GATE				C/T M1 M0		0	0	MCS-48 Timer. "TLx" serves as five-bit prescaler.
		TIMER 1				TIMER 0		0	1	16-bit timer/counter. "THx" and "TLx" are cascaded; there is no prescaler.
								1	0	8-bit auto-reload timer/counter. "THx" holds a value which is to be reloaded into "TLx" each time it overflows.
GATE	Gating control. When set, Timer/counter "x" is enabled only while "INTx" pin is high and "TRx" control bit is set. When cleared, timer/counter is enabled whenever "TRx" control bit is set.						1	1	(Timer 0) TL0 is an eight-bit timer/counter controlled by the standard Timer 0 control bits.	
C/T	Timer or Counter Selector. Cleared for Timer operation (input from internal system clock). Set for Counter operation (input from "Tx" input pin).						1	1	TH0 is an eight-bit timer only controlled by Timer 1 control bits.	
										(Timer 1) Timer/counter 1 stopped.

Figure 7. TMOD—Timer/Counter Mode Register

	(MSB)									(LSB)
	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0		
<b>Symbol Position Name and Significance</b>										
TF1	TCON.7	Timer 1 overflow Flag. Set by hardware on timer/counter overflow. Cleared when interrupt processed.			IE1	TCON.3	Interrupt 1 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.			
TR1	TCON.6	Timer 1 Run control bit. Set/cleared by software to turn timer/counter on/off.			IT1	TCON.2	Interrupt 1 Type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts.			
TF0	TCON.5	Timer 0 overflow Flag. Set by hardware on timer/counter overflow. Cleared when interrupt processed.			IE0	TCON.1	Interrupt 0 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.			
TR0	TCON.4	Timer 0 Run control bit. Set/cleared by software to turn timer/counter on/off.			IT0	TCON.0	Interrupt 0 Type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts.			

Figure 8. TCON—Timer/Counter Control/Status Register



Symbol	Position	Name and Significance
SM0	SCON.7	Serial port Mode control bit 0. Set/cleared by software (see note).
SM1	SCON.6	Serial port Mode control bit 1. Set/cleared by software (see note).
SM2	SCON.5	Serial port Mode control bit 2. Set by software to disable reception of frames for which bit 8 is zero.
REN	SCON.4	Receiver Enable control bit. Set/cleared by software to enable/disable serial data reception.
TB8	SCON.3	Transmit Bit 8. Set/cleared by hardware to determine state of ninth data bit transmitted in 9-bit UART mode.

Symbol	Position	Name and Significance
RB8	SCON.2	Receive Bit 8. Set/cleared by hardware to indicate state of ninth data bit received.
TI	SCON.1	Transmit Interrupt flag. Set by hardware when byte transmitted. Cleared by software after servicing.
RI	SCON.0	Received Interrupt flag. Set by hardware when byte received. Cleared by software after servicing.

**Note—** the state of (SM0,SM1) selects:  
 (0,0)—Shift register I/O expansion.  
 (0,1)—8 bit UART, variable data rate.  
 (1,0)—9 bit UART, fixed data rate.  
 (1,1)—9 bit UART, variable data rate.

Figure 9. SCON—Serial Port Control/Status Register

These peripheral functions allow special hardware to monitor real-time signal interfacing without bothering the CPU. For example, imagine serial data is arriving from one CRT while being transmitted to another, and one timer/counter is tallying high-speed input transitions while the other measures input pulse widths. During all of this the CPU is thinking about something else.

But how does the CPU know when a reception, transmission, count, or pulse is finished? The 8051 programmer can choose from three approaches.

TCON and SCON contain status bits set by the hardware when a timer overflows or a serial port operation is completed. The first technique reads the control register into the accumulator, tests the appropriate bit, and does a conditional branch based on the result. This “polling” scheme (typically a three-instruction sequence though additional instructions to save and restore the accumulator may sometimes be needed) will surely be familiar to programmers used to multi-chip microcomputer systems and peripheral controller chips. This process is rather cumbersome, especially when monitoring multiple peripherals.

As a second approach, the 8051 can perform a conditional branch based on the state of any control or status bit or input pin in a single instruction; a four instruction sequence could poll the four simultaneous happenings mentioned above in just eight microseconds.

Unfortunately, the CPU must still drop what it's doing to test these bits. A manager cannot do his own work well if he is continuously monitoring his subordinates; they should interrupt him (or her) only when they need attention or guidance. So it is with machines: ideally, the CPU would not have to worry about the peripherals until they require servicing. At that time, it would postpone the

background task long enough to handle the appropriate device, then return to the point where it left off.

This is the basis of the third and generally optimal solution, hardware interrupts. The 8051 has five interrupt sources: one from the serial port when a transmission or reception is complete, two from the timers when overflows occur, and two from input pins INT0 and INT1. Each source may be independently enabled or disabled to allow polling on some sources or at some times, and each may be classified as high or low priority. A high priority source can interrupt a low priority service routine; the manager's boss can interrupt conferences with subordinates. These options are selected by the interrupt enable and priority control registers, IE and IP (Figures 10 and 11).

Each source has a particular program memory address associated with it (Table 3), starting at 0003H (as in the 8048) and continuing at eight-byte intervals. When an event enabled for interrupts occurs the CPU automatically executes an internal subroutine call to the corresponding address. A user subroutine starting at this location (or jumped to from this location) then performs the instructions to service that particular source. After completing the interrupt service routine, execution returns to the background program.

Table 3. 8051 Interrupt Sources and Service Vectors

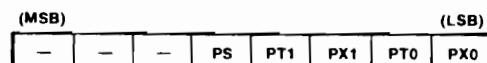
Interrupt Source	Service Routine Starting Address
(Reset)	0000H
External 0	0003H
Timer/Counter 0	000BH
External 1	0013H
Timer/Counter 1	001BH
Serial Port	0023H



Symbol Position		Name and Significance
EA	IE.7	Enable All control bit. Cleared by software to disable all interrupts, independent of the state of IE.4-IE.0.
-	IE.6	(reserved)
-	IE.5	(reserved)
ES	IE.4	Enable Serial port control bit. Set/cleared by software to enable/disable interrupts from TI or RI flags.
ET1	IE.3	Enable Timer 1 control bit. Set/cleared by software to enable/disable interrupts from timer/counter 1.

Symbol Position		Name and Significance
EX1	IE.2	Enable External interrupt 1 control bit. Set/cleared by software to enable/disable interrupts from INT1.
ET0	IE.1	Enable Timer 0 control bit. Set/cleared by software to enable/disable interrupts from timer/counter 0
EX0	IE.0	Enable External interrupt 0 control bit. Set/cleared by software to enable/disable interrupts from INT0.

Figure 10. IE—Interrupt Enable Register



Symbol Position		Name and Significance
-	IP.7	(reserved)
-	IP.6	(reserved)
-	IP.5	(reserved)
PS	IP.4	Serial port Priority control bit. Set/cleared by software to specify high/low priority interrupts for Serial port.
PT1	IP.3	Timer 1 Priority control bit. Set/cleared by software to specify high/low priority interrupts for timer/counter 1.

Symbol Position		Name and Significance
PX1	IP.2	External interrupt 1 Priority control bit. Set/cleared by software to specify high/low priority interrupts for INT1.
PT0	IP.1	Timer 0 Priority control bit. Set/cleared by software to specify high/low priority interrupts for timer/counter 0.
PX0	IP.0	External interrupt 0 Priority control bit. Set/cleared by software to specify high/low priority interrupts for INT0.

Figure 11. IP—Interrupt Priority Control Register

Table 4. MCS-51™ Instruction Set Description

ARITHMETIC OPERATIONS			DATA TRANSFER (cont.)		
Mnemonic	Description	Byte Cyc	Mnemonic	Description	Byte Cyc
ADD A.Rn	Add register to Accumulator	1 1	MOVC A.@A+DPTR	Move Code byte relative to DPTR to A	1 2
ADD A.direct	Add direct byte to Accumulator	2 1	MOVC A.@A+PC	Move Code byte relative to PC to A	1 2
ADD A.@Ri	Add indirect RAM to Accumulator	1 1	MOVX A,@Ri	Move External RAM (8-bit addr) to A	1 2
ADD A.#data	Add immediate data to Accumulator	2 1	MOVX A.@DPTR	Move External RAM (16-bit addr) to A	1 2
ADDC A.Rn	Add register to Accumulator with Carry	1 1	MOVX @Ri,A	Move A to External RAM (8-bit addr)	1 2
ADDC A.direct	Add direct byte to A with Carry flag	2 1	MOVX @DPTR,A	Move A to External RAM (16-bit addr)	1 2
ADDC A.@Ri	Add indirect RAM to A with Carry flag	1 1	PUSH direct	Push direct byte onto stack	2 2
ADDC A.#data	Add immediate data to A with Carry flag	2 1	POP direct	Pop direct byte from stack	2 2
SUBB A.Rn	Subtract register from A with Borrow	1 1	XCH A.Rn	Exchange register with Accumulator	1 1
SUBB A.direct	Subtract direct byte from A with Borrow	2 1	XCH A.direct	Exchange direct byte with Accumulator	2 1
SUBB A.@Ri	Subtract indirect RAM from A w/ Borrow	1 1	XCH A.@Ri	Exchange indirect RAM with A	1 1
SUBB A.#data	Subtract immmed. data from A w/ Borrow	2 1	XCHD A.@Ri	Exchange low-order Digit ind. RAM w/ A	1 1
INC A	Increment Accumulator	1 1	BOOLEAN VARIABLE MANIPULATION		
INC Rn	Increment register	1 1	Mnemonic	Description	Byte Cyc
INC direct	Increment direct byte	2 1	CLR C	Clear Carry flag	1 1
INC @Ri	Increment indirect RAM	1 1	CLR bit	Clear direct bit	2 1
DEC A	Decrement Accumulator	1 1	SETB C	Set Carry flag	1 1
DEC Rn	Decrement register	1 1	SETB bit	Set direct Bit	2 1
DEC direct	Decrement direct byte	2 1	CPL C	Complement Carry flag	1 1
DEC @Ri	Decrement indirect RAM	1 1	CPL bit	Complement direct bit	2 1
INC DPTR	Increment Data Pointer	1 2	ANL C.bit	AND direct bit to Carry flag	2 2
MUL AB	Multiply A & B	1 4	ANL C./bit	AND complement of direct bit to Carry	2 2
DIV AB	Divide A by B	1 4	ORL C.bit	OR direct bit to Carry flag	2 2
DA A	Decimal Adjust Accumulator	1 1	ORL C./bit	OR complement of direct bit to Carry	2 2
LOGICAL OPERATIONS			MOV C.bit	Move direct bit to Carry flag	2 1
Mnemonic	Destination	Byte Cyc	MOV bit,C	Move Carry flag to direct bit	2 2
			PROGRAM AND MACHINE CONTROL		
ANL A.Rn	AND register to Accumulator	1 1	Mnemonic	Description	Byte Cyc
ANL A.direct	AND direct byte to Accumulator	2 1	ACALL addr11	Absolute Subroutine Call	2 2
ANL A.@Ri	AND indirect RAM to Accumulator	1 1	LCALL addr16	Long Subroutine Call	3 2
ANL A.#data	AND immediate data to Accumulator	2 1	RET	Return from subroutine	1 2
ANL direct,A	AND Accumulator to direct byte	2 1	RETI	Return from interrupt	1 2
ANL direct,#data	AND immediate data to direct byte	3 2	AJMP addr11	Absolute Jump	2 2
ORL A.Rn	OR register to Accumulator	1 1	LJMP addr16	Long Jump	3 2
ORL A.direct	OR direct byte to Accumulator	2 1	SJMP rel	Short Jump (relative addr)	2 2
ORL A.@Ri	OR indirect RAM to Accumulator	1 1	JMP @A+DPTR	Jump indirect relative to the DPTR	1 2
ORL A.#data	OR immediate data to Accumulator	2 1	JZ rel	Jump if Accumulator is Zero	2 2
ORL direct,A	OR Accumulator to direct byte	2 1	JNZ rel	Jump if Accumulator is Not Zero	2 2
ORL direct,#data	OR immediate data to direct byte	3 2	JC rel	Jump if Carry flag is set	2 2
XRL A.Rn	Exclusive-OR register to Accumulator	1 1	JNC rel	Jump if No Carry flag	2 2
XRL A.direct	Exclusive-OR direct byte to Accumulator	2 1	JB bit,rel	Jump if direct Bit set	3 2
XRL A.@Ri	Exclusive-OR indirect RAM to A	1 1	JNB bit,rel	Jump if direct Bit Not set	3 2
XRL A.#data	Exclusive-OR immediate data to A	2 1	JBC bit,rel	Jump if direct Bit is set & Clear bit	3 2
XRL direct,A	Exclusive-OR Accumulator to direct byte	2 1	CJNE A.direct,rel	Compare direct to A & Jump if Not Equal	3 2
XRL direct,#data	Exclusive-OR immediate data to direct	3 2	CJNE A.#data,rel	Comp. immmed. to A & Jump if Not Equal	3 2
CLR A	Clear Accumulator	1 1	CJNE Rn,#data,rel	Comp. immmed. to reg. & Jump if Not Equal	3 2
CPL A	Complement Accumulator	1 1	CJNE @Ri,#data,rel	Comp. immmed. to ind. & Jump if Not Equal	3 2
RL A	Rotate Accumulator Left	1 1	DJNZ Rn,rel	Decrement register & Jump if Not Zero	2 2
RLC A	Rotate A Left through the Carry flag	1 1	DJNZ direct,rel	Decrement direct & Jump if Not Zero	3 2
RR A	Rotate Accumulator Right	1 1	NOP	No operation	1 1
RRC A	Rotate A Right through Carry flag	1 1	Notes on data addressing modes:		
SWAP A	Swap nibbles within the Accumulator	1 1	Rn	—Working register R0-R7	
DATA TRANSFER			direct	—128 internal RAM locations, any I/O port, control or status register	
Mnemonic	Description	Byte Cyc	@Ri	—Indirect internal RAM location addressed by register R0 or R1	
MOV A.Rn	Move register to Accumulator	1 1	#data	—8-bit constant included in instruction	
MOV A.direct	Move direct byte to Accumulator	2 1	#data16	—16-bit constant included as bytes 2 & 3 of instruction	
MOV A.@Ri	Move indirect RAM to Accumulator	1 1	bit	—128 software flags, any I/O pin, control or status bit	
MOV A.#data	Move immediate data to Accumulator	2 1	Notes on program addressing modes:		
MOV Rn,A	Move Accumulator to register	1 1	addr16	—Destination address for LCALL & LJMP may be anywhere within the 64-Kilobyte program memory address space.	
MOV Rn,direct	Move direct byte to register	2 2	addr11	—Destination address for ACALL & AJMP will be within the same 2-Kilobyte page of program memory as the first byte of the following instruction.	
MOV Rn,#data	Move immediate data to register	2 1	rel	—SJMP and all conditional jumps include an 8-bit offset byte. Range is +127/-128 bytes relative to first byte of the following instruction.	
MOV direct,A	Move Accumulator to direct byte	2 1	All mnemonics copyrighted © Intel Corporation 1979		
MOV direct,Rn	Move register to direct byte	2 2			
MOV direct,direct	Move direct byte to direct	3 2			
MOV direct,@Ri	Move indirect RAM to direct byte	2 2			
MOV direct,#data	Move immediate data to direct byte	3 2			
MOV @Ri,A	Move Accumulator to indirect RAM	1 1			
MOV @Ri,direct	Move direct byte to indirect RAM	2 2			
MOV @Ri,#data	Move immediate data to indirect RAM	2 1			
MOV DPTR,#data16	Load Data Pointer with a 16-bit constant	3 2			

### 3. INSTRUCTION SET AND ADDRESSING MODES

The 8051 instruction set is extremely regular, in the sense that most instructions can operate with variables from several different physical or logical address spaces. Before getting deeply enmeshed in the instruction set proper, it is important to understand the details of the most common data addressing modes. Whereas Table 4 summarizes the instructions set broken down by functional

group, this chapter starts with the addressing mode classes and builds to include the related instructions.

#### Data Addressing Modes

MCS-51 assembly language instructions consist of an operation mnemonic and zero to three operands separated by commas. In two operand instructions the destination is specified first, then the source. Many byte-wide data

operations (such as ADD or MOV) inherently use the accumulator as a source operand and/or to receive the result. For the sake of clarity the letter "A" is specified in the source or destination field in all such instructions. For example, the instruction,

ADD A,<source>

will add the variable <source> to the accumulator, leaving the sum in the accumulator.

The operand designated "<source>" above may use any of four common logical addressing modes:

- Register—one of the working registers in the currently enabled bank.
- Direct—an internal RAM location, I/O port, or special-function register.
- Register-indirect—an internal RAM location, pointed to by a working register.
- Immediate data—an eight-bit constant incorporated into the instruction.

The first three modes provide access to the internal RAM and Hardware Register address spaces, and may therefore be used as source or destination operands; the last mode accesses program memory and may be a source operand only.

(It is hard to show a "typical application" of any instruction without involving instructions not yet described. The following descriptions use only the self-explanatory ADD and MOV instructions to demonstrate how the four addressing modes are specified and used. Subsequent examples will become increasingly complex.)

### Register Addressing

The 8051 programmer has access to eight "working registers," numbered R0-R7. The least-significant three-bits of the instruction opcode indicate one register within this logical address space. Thus, a function code and operand address can be combined to form a short (one byte) instruction (Figure 12.a).

The 8051 assembly language indicates register addressing with the symbol Rn (where n is from 0 to 7) or with a symbolic name previously defined as a register by the EQUATE or SET directives. (For more information on assembler directives see the Macro Assembler Reference Manual.)

### Example 1—Adding Two Registers Together

```
; REGADR ADD CONTENTS OF REGISTER 1
; TO CONTENTS OF REGISTER 0
; 
REGADR MOV A, R0
ADD A, R1
MOV R0, A
```

There are four such banks of working registers, only one of which is active at a time. Physically, they occupy the first 32 bytes of on-chip data RAM (addresses 0-1FH). PSW bits 4 and 3 determine which bank is active. A

hardware reset enables register bank 0; to select a different bank the programmer modifies PSW bits 4 and 3 accordingly.

### Example 2—Selecting Alternate Memory Banks

```
MOV PSW, #0001000B ; SELECT BANK 2
```

Register addressing in the 8051 is the same as in the 8048 family, with two enhancements: there are four banks rather than one or two, and 16 instructions (rather than 12) can access them.

### Direct Byte Addressing

Direct addressing can access any on-chip variable or hardware register. An additional byte appended to the opcode specifies the location to be used (Figure 12.b).

Depending on the highest order bit of the direct address byte, one of two physical memory spaces is selected. When the direct address is between 0 and 127 (00H-7FH) one of the 128 low-order on-chip RAM locations is used. (Future microcomputers based on the MCS-51™ architecture may incorporate more than 128 bytes of on-chip RAM. Even if this is the case, only the low-order 128 bytes will be directly addressable. The remainder would be accessed indirectly or via the stack pointer.)

### Example 3—Adding RAM Location Contents

```
; DIRADR ADD CONTENTS OF RAM LOCATION 41H
; TO CONTENTS OF RAM LOCATION 40H
; 
DIRADR MOV A, 40H
ADD A, 41H
MOV 40H, A
```

All I/O ports and special function, control, or status registers are assigned addresses between 128 and 255 (80H-0FFH). When the direct address byte is between these limits the corresponding hardware register is accessed. For example, Ports 0 and 1 are assigned direct addresses 80H and 90H, respectively. A complete list is presented in Table 5. Don't waste your time trying to memorize the addresses in Table 5. Since programs using absolute addresses for function registers would be difficult to write or understand, ASM51 allows and understands the abbreviations listed instead.

### Example 4—Adding Input Port Data to Output Port Data

```
; PRTADR ADD DATA INPUT ON PORT 1
; TO DATA PREVIOUSLY OUTPUT
; ON PORT 0
; 
PRTADR MOV A, P0
ADD A, P1
MOV P0, A
```

Direct addressing allows all special-function registers in the 8051 to be read, written, or used as instruction operands. In general, this is the *only* method used for accessing I/O ports and special-function registers. If direct addressing is used with special-function register addresses other than those listed, the result of the instruction is undefined.

The 8048 does not have or need any generalized direct addressing mode, since there are only five special registers (BUS, P1, P2, PSW, & T) rather than twenty. Instead, 16 special 8048 opcodes control output bits or read or write each register to the accumulator. These functions are all subsumed by four of the 27 direct addressing instructions of the 8051.

**Table 5. 8051 Hardware Register Direct Addresses**

Register	Address	Function
P0	80H*	Port 0
SP	81H	Stack Pointer
DPL	82H	Data Pointer (Low)
DPH	83H	Data Pointer (High)
TCON	88H*	Timer register
TMOD	89H	Timer Mode register
TL0	8AH	Timer 0 Low byte
TL1	8BH	Timer 1 Low byte
TH0	8CH	Timer 0 High byte
TH1	8DH	Timer 1 High byte
PI	90H*	Port 1
SCON	98H*	Serial Port Control register
SBUF	99H	Serial Port data Buffer
P2	0A0H*	Port 2
IE	0A8H*	Interrupt Enable register
P3	0B0H*	Port 3
IP	0B8H*	Interrupt Priority register
PSW	0D0H*	Program Status Word
ACC	0E0H*	Accumulator (direct address)
B	0F0H*	B register

\* = bit addressable register.

### Register-Indirect Addressing

How can you handle variables whose locations in RAM are determined, computed, or modified while the program is running? This situation arises when manipulating sequential memory locations, indexed entries within tables in RAM, and multiple precision or string operations. Register or Direct addressing cannot be used, since their operand addresses are fixed at assembly time.

The 8051 solution is "register-indirect RAM addressing." R0 and R1 of each register bank may operate as index or pointer registers, their contents indicating an address into RAM. The internal RAM location so addressed is the actual operand used. The least significant bit of the instruction opcode determines which register is used as the "pointer" (Figure 12.c).

In the 8051 assembly language, register-indirect addressing is represented by a commercial "at" sign ("@") preceding R0, R1, or a symbol defined by the user to be equal to R0 or R1.

### Example 5—Indirect Addressing

```

    ; INADR ADD CONTENTS OF MEMORY LOCATION
    ;          ADDRESSED BY REGISTER 1
    ;          TO CONTENTS OF RAM LOCATION
    ;          ADDRESSED BY REGISTER 0

    INADR MOV    A, @R0
    ADD     A, @R1
    MOV    R0, A
  
```

Indirect addressing on the 8051 is the same as in the 8048 family, except that all eight bits of the pointer register contents are significant; if the contents point to a non-existent memory location (i.e., an address greater than 7FH on the 8051) the result of the instruction is undefined. (Future microcomputers based on the MCS-51™ architecture could implement additional memory in the on-chip RAM logical address space at locations above 7FH.) The 8051 uses register-indirect addressing for five new instructions plus the 13 on the 8048.

### Immediate Addressing

When a source operand is a constant rather than a variable (i.e.—the instruction uses a value known at assembly time), then the constant can be incorporated into the instruction. An additional instruction byte specifies the value used (Figure 12.d).

The value used is fixed at the time of ROM manufacture or EPROM programming and may not be altered during program execution. In the assembly language immediate operands are preceded by a number sign ("#"). The operand may be either a numeric string, a symbolic variable, or an arithmetic expression using constants.

### Example 6—Adding Constants Using Immediate Addressing

```

    ; IMMADR ADD THE CONSTANT 12 (DECIMAL)
    ;          TO THE CONSTANT 34 (DECIMAL).
    ;          LEAVE SUM IN ACCUMULATOR.

    IMMADR MOV    A, #12
    ADD     A, #34
  
```

The preceding example was included for consistency; it has little practical value. Instead, ASM51 could compute the sum of two constants at assembly time.

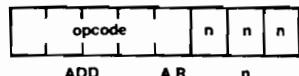
### Example 7—Adding Constants Using ASM51 Capabilities

```

    ; ASMSUM LOAD ACC WITH THE SUM OF
    ;          THE CONSTANT 12 (DECIMAL) AND
    ;          THE CONSTANT 34 (DECIMAL).

    ASMSUM: MOV    A, #(12+34)
  
```

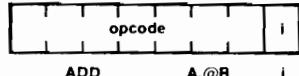
#### a.) Register Addressing:



#### b.) Direct Addressing:



#### c.) Register-indirect Addressing:



#### d.) Immediate Addressing:



**Figure 12. Data Addressing Machine Code Formats**

### Addressing Mode Combinations

The above examples all demonstrated the use of the four data-addressing modes in two-operand instructions (MOV, ADD) which use the accumulator as one operand. The operations ADDC, SUBB, ANL, ORL, and XRL (all to be discussed later) could be substituted for ADD in each example. The first three modes may be also be used for the XCH operation or, in combination with the Immediate Addressing mode (and an additional byte), loaded with a constant. The one-operand instructions INC and DEC, DJNZ, and CJNE may all operate on the accumulator, or may specify the Register, Direct, and Register-indirect addressing modes. Exception: as in the 8048, DJNZ cannot use the accumulator or indirect addressing. (The PUSH and POP operations cannot inherently address the accumulator as a special register either. However, all three can *directly* address the accumulator as one of the twenty special-function registers by putting the symbol "ACC" in the operand field.)

### Advantages of Symbolic Addressing

Like most assembly or higher-level programming languages, ASM51 allows instructions or variables to be given appropriate, user-defined symbolic names. This is done for instruction lines by putting a label followed by a colon (":") before the instruction proper, as in the above examples. Such symbols must start with an alphabetic character (remember what distinguished BACH from 0BACH?), and may include any combination of letters, numbers, question marks ("?") and underscores ("\_"). For very long names only the first 31 characters are relevant.

Assembly language programs may intermix upper- and lower-case letters arbitrarily, but ASM51 converts both to upper-case. For example, ASM51 will internally process an "l" for an "i" and, of course, "A\_TOOTH" for "a\_tooth."

The underscore character makes symbols easier to read and can eliminate potential ambiguity (as in the label for a subroutine to switch two entires on a stack, "S\_EXCHANGE"). The underscore is significant, and would distinguish between otherwise-identical character strings.

ASM51 allows *all* variables (registers, ports, internal or external RAM addresses, constants, etc.) to be assigned labels according to these rules with the EQUate or SET directives.

#### Example 8—Symbolic Addressing of Variables Defined as RAM Locations

```

VAR_0  SET    20H
VAR_1  SET    21H
;
SYMB_1 ADD CONTENTS OF VAR_1
          TO CONTENTS OF VAR_0
;
SYMB_1: MOV    A, VAR_0
        ADD    A, VAR_1
        MOV    VAR_0, A

```

Notice from Table 4 that the MCS-51™ instruction set has relatively few instruction mnemonics (abbreviations) for the programmer to memorize. Different data types or addressing modes are determined by the operands specified, rather than variations on the mnemonic. For example, the mnemonic "MOV" is used by 18 different instructions to operate on three data types (bit, byte, and address). The fifteen versions which move byte variables between the logical address spaces are diagrammed in Figure 13. Each arrow shows the direction of transfer from source to destination.

Notice also that for most instructions allowing register addressing there is a corresponding direct addressing instruction and vice versa. This lets the programmer begin writing 8051 programs as if (s)he has access to 128 different registers. When the program has evolved to the point where the programmer has a fairly accurate idea how often each variable is used, he/she may allocate the working registers in each bank to the most "popular" variables. (The assembly cross-reference option will show exactly how often and where each symbol is referenced.) If symbolic addressing is used in writing the source program only the lines containing the symbol definition will need to be changed; the assembler will produce the appropriate instructions even though the rest of the program is left untouched. Editing only the first two lines of Example 8 will shrink the six-byte code segment produced in half.

How are instruction sets "counted"? There is no standard practice; different people assessing the same CPU using different conventions may arrive at different totals.

Each operation is then broken down according to the different addressing modes (or combinations of addressing modes) it can accommodate. The "CLR" mnemonic is used by two instructions with respect to bit variables ("CLR C" and "CLR bit") and once ("CLR A") with regards to bytes. This expansion yields the 111 separate instructions of Table 4.

The method used for the MCS-51® instruction set first breaks it down into "operations": a basic function applied to a single data type. For example, the four versions of the ADD instruction are grouped to form one operation — addition of eight-bit variables. The six forms of the ANL instruction for byte variables make up a different operation; the two forms of ANL which operate on *bits* are considered still another. The MOV mnemonic is used by three different operation classes, depending on whether bit, byte, or 16-bit values are affected. Using this terminology the 8051 can perform 51 different operations.

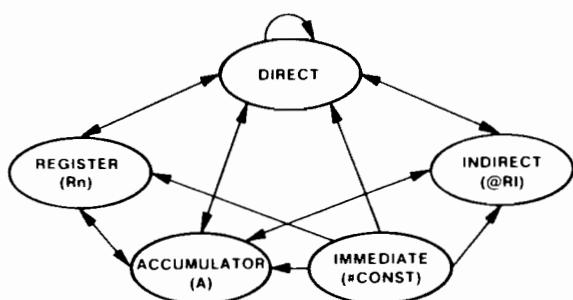


Figure 13. Road map for moving data bytes

**Example 9—Redeclaring Example 8 Symbols as Registers**

```

; VAR_0  SET      R0
; VAR_1  SET      R1
;
; SYMB_2 ADD CONTENTS OF VAR_1
;           TO CONTENTS OF VAR_0
;
SYMB_2: MOV      A, VAR_0
        ADD      A, VAR_1
        MOV      VAR_0, A

```

**Arithmetic Instruction Usage — ADD, ADDC, SUBB and DA**

The ADD instruction adds a byte variable with the accumulator, leaving the result in the accumulator. The carry flag is set if there is an overflow from bit 7 and cleared otherwise. The AC flag is set to the carry-out from bit 3 for use by the DA instruction described later. ADDC adds the previous contents of the carry flag with the two byte variables, but otherwise is the same as ADD.

The SUBB (subtract with borrow) instruction subtracts the byte variable indicated and the contents of the carry flag together from the accumulator, and puts the result back in the accumulator. The carry flag serves as a "Borrow Required" flag during subtraction operations; when a greater value is subtracted from a lesser value (as in subtracting 5 from 1) requiring a borrow into the highest order bit, the carry flag is set; otherwise it is cleared.

When performing signed binary arithmetic, certain combinations of input variables can produce results which seem to violate the Laws of Mathematics. For example, adding 7FH (127) to itself produces a sum of 0FEH, which is the two's complement representation of -2 (refer back to Table 2)! In "normal" arithmetic, two positive values can't have a negative sum. Similarly, it is normally impossible to subtract a positive value from a negative value and leave a positive result — but in two's complement there are instances where this too may happen. Fundamentally, such anomalies occur when the magnitude of the resulting value is too great to "fit" into the seven bits allowed for it; there is no one-byte two's complement representation for 254, the true sum of 127 and 127.

The MCS-51™ processors detect whether these situations occur and indicate such errors with the OV flag. (OV may be tested with the conditional jump instructions JB and JNB, described under the Boolean Processor chapter.)

At a hardware level, OV is set if there is a carry out of bit 6 but not out of bit 7, or a carry out of bit 7 but not out of bit 6. When adding signed integers this indicates a negative number produced as the sum of two positive operands, or a positive sum from two negative operands; on SUBB this indicates a negative result after subtracting a negative number from a positive number, or a positive result when a positive number is subtracted from a negative number.

The ADDC and SUBB instructions incorporate the previous state of the carry (borrow) flag to allow multiple precision calculations by repeating the operation with successively higher-order operand bytes. In either case, the carry must be cleared before the first iteration.

If the input data for a multiple precision operation is an unsigned string of integers, upon completion the carry flag will be set if an overflow (for ADDC) or underflow (for SUBB) occurs. With two's complement signed data (i.e., if the most significant bit of the original input data indicates the sign of the string), the overflow flag will be set if overflow or underflow occurred.

**Example 10—String Subtraction with Signed Overflow Detection**

```

; SUBSTR SUBTRACT STRING INDICATED BY RI
;           FROM STRING INDICATED BY R0 TO
;           PRECISION INDICATED BY R2
;           CHECK FOR SIGNED UNDERFLOW WHEN DONE.
;
SUBSTR: CLR      C, .BORROW= 0.
SUBS1: MOV      A, R0
        SUBB    A, RI, .SUBTRACT NEXT PLACE
        MOV      R0, A, .BUMP POINTERS
        INC      RI
        INC
        DJNZ    R2, SUBS1, .LOOP AS NEEDED
        ; WHEN DONE, TEST IF OVERFLOW OCCURED
        ; ON LAST ITERATION OF LOOP
        JNB      OV, OV_OK, .OVERFLOW RECOVERY ROUTINE
        OV_OK: RET, .RETURN

```

Decimal addition is possible by using the DA instruction in conjunction with ADD and/or ADDC. The eight-bit binary value in the accumulator resulting from an earlier addition of two variables (each a packed BCD digit-pair) is adjusted to form two BCD digits of four bits each. If the contents of accumulator bits 3-0 are greater than nine (xxxx1010-xxxx1111), or if the AC flag had been set, six is added to the accumulator producing the proper BCD digit in the low-order nibble. (This addition might itself set — but would not clear — the carry flag.) If the carry flag is set, or if the four high-order bits now exceed nine (1010xxxx-1111xxxx), these bits are incremented by six. The carry flag is left set if originally set or if either addition of six produces a carry out of the highest-order bit, indicating the sum of the original two BCD variables is greater than or equal to decimal 100.

### Example 11—Two Byte Decimal Add with Registers and Constants

```
; BCDADD ADD THE CONSTANT 1,234 (DECIMAL) TO THE
; CONTENTS OF REGISTER PAIR <R3><R2>
; (ALREADY A 4 BCD-DIGIT VARIABLE)
;
BCDADD. MOV A, R2
ADD A, #34H
DA A
MOV R2, A
MOV A, R3
ADDC A, #12H
DA A
MOV R3, A
RET
```

### Multiplication and Division

The instruction "MUL AB" multiplies the unsigned eight-bit integer values held in the accumulator and B-registers. The low-order byte of the sixteen-bit product is left in the accumulator, the higher-order byte in B. If the high-order eight-bits of the product are all zero the overflow flag is cleared; otherwise it is set. The programmer can poll OV to determine when the B register is non-zero and must be processed.

"DIV AB" divides the unsigned eight-bit integer in the accumulator by the unsigned eight-bit integer in the B-register. The integer part of the quotient is returned in the accumulator; the remainder in the B-register. If the B-register originally contained 00H then the overflow flag will be set to indicate a division error, and the values returned will be undefined. Otherwise OV is cleared.

The divide instruction is also useful for purposes such as radix conversion or separating bit fields of the accumulator. A short subroutine can convert an eight-bit unsigned binary integer in the accumulator (between 0 & 255) to a three-digit (two byte) BCD representation. The hundred's digit is returned in one register (HUND) and the ten's and one's digits returned as packed BCD in another (TENONE).

### Example 12—Use of DIV Instruction for Radix Conversion

```
; BINBCD CONVERT 8-BIT BINARY VARIABLE IN ACC
; TO 3-DIGIT PACKED BCD FORMAT
; HUNDREDS' PLACE LEFT IN VARIABLE 'HUND'.
; TENS' AND ONES' PLACES IN 'TENONE'.
;
HUND EQU 21H
TENONE EQU 22H
;
BINBCD. MOV B, #100 ; DIVIDE BY 100 TO
DIV AB ; DETERMINE NUMBER OF HUNDREDS
MOV HUND, A
MOV A, #10 ; DIVIDE REMAINDER BY 10 TO
XCH A, B ; DETERMINE # OF TENS LEFT
DIV AB ; TENS DIGIT IN ACC. REMAINDER IS ONES
; DIGIT
SWAP A
ADD A, B ; PACK BCD DIGITS IN ACC
MOV TENONE, A
RET
```

The divide instruction can also separate eight bits of data in the accumulator into sub-fields. For example, packed BCD data may be separated into two nibbles by dividing the data by 16, leaving the high-nibble in the accumulator and the low-order nibble (remainder) in B. The two digits may then be operated on individually or in conjunction with each other. This example receives two packed BCD

digits in the accumulator and returns the product of the two individual digits in packed BCD format in the accumulator.

### Example 13—Implementing a BCD Multiply Using MPY and DIV

```
; MULBCD UNPACK TWO BCD DIGITS RECEIVED IN ACC.
; FIND THEIR PRODUCT, AND RETURN PRODUCT
; IN PACKED BCD FORMAT IN ACC.
;
MULBCD. MOV B, #10H ; DIVIDE INPUT BY 16
DIV AB ; A & B HOLD SEPARATED DIGITS
; (EACH RIGHT JUSTIFIED IN REGISTER)
MUL AB ; A HOLDS PRODUCT IN BINARY FORMAT (0 -
; 99 (DECIMAL) = D - 63H)
MOV B, #10 ; DIVIDE PRODUCT BY 10
DIV AB ; A HOLDS # OF TENS. B HOLDS REMAINDER
SWAP A
ORL A, B ; PACK DIGITS
RET
```

### Logical Byte Operations — ANL, ORL, XRL

The instructions ANL, ORL, and XRL perform the logical functions AND, OR, and/or Exclusive-OR on the two byte variables indicated, leaving the results in the first. No flags are affected. (A word to the wise — do not vocalize the first two mnemonics in mixed company.)

These operations may use all the same addressing modes as the arithmetics (ADD, etc.) but unlike the arithmetics, they are not restricted to operating on the accumulator. Directly addressed bytes may be used as the destination with either the accumulator or a constant as the source. These instructions are useful for clearing (ANL), setting (ORL), or complementing (XRL) one or more bits in a RAM, output ports, or control registers. The pattern of bits to be affected is indicated by a suitable mask byte. Use immediate addressing when the pattern to be affected is known at assembly time (Figure 14); use the accumulator versions when the pattern is computed at run-time.

I/O ports are often used for parallel data in formats other than simple eight-bit bytes. For example, the low-order five bits of port 1 may output an alphabetic character code (hopefully) without disturbing bits 7-5. This can be a simple two-step process. First, clear the low-order five pins with an ANL instruction; then set those pins corresponding to ones in the accumulator. (This example assumes the three high-order bits of the accumulator are originally zero.)

### Example 14—Reconfiguring Port Size with Logical Byte Instructions

```
OUT_PX: ANL P1, #1110000B ; CLEAR BITS P1.4 - P1.0
ORL P1, A ; SET P1 PINS CORRESPONDING TO SET ACC
; BITS
RET
```

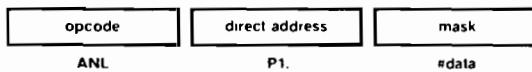


Figure 14. Instruction Pattern for Logical Operation Special Addressing Modes

In this example, low-order bits remaining high may "glitch" low for one machine cycle. If this is undesirable, use a slightly different approach. First, set all pins corresponding to accumulator one bits, then clear the pins corresponding to zeroes in low-order accumulator bits. Not all bits will change from original to final state at the same instant, but no bit makes an intermediate transition.

**Example 15—Reconfiguring I/O Port Size without Glitching**

```
ALT_PX ORL
      ORL
      ANL
      RET
```

P1.A  
A. #11100000B  
P1.A

**Program Control — Jumps, Calls, Returns**

Whereas the 8048 only has a single form of the simple jump instruction, the 8051 has three. Each causes the program to unconditionally jump to some other address. They differ in how the machine code represents the destination address.

LJMP (Long Jump) encodes a sixteen-bit address in the second and third instruction bytes (Figure 15.a); the destination may be anywhere in the 64 Kilobyte program memory address space.

The two-byte AJMP (Absolute Jump) instruction encodes its destination using the same format as the 8048: address bits 10 through 8 form a three bit field in the opcode and address bits 7 through 0 form the second byte (Figure 15.b). Address bits 15-12 are unchanged from the (incremented) contents of the P.C., so AJMP can only be used when the destination is known to be within the same 2K memory block. (Otherwise ASM51 will point out the error.)

A different two-byte jump instruction is legal with any nearby destination, regardless of memory block boundaries or "pages." SJMP (Short Jump) encodes the destination with a program counter-relative address in the second byte (Figure 15.c). The CPU calculates the

destination at run-time by adding the signed eight-bit displacement value to the incremented P.C. Negative offset values will cause jumps up to 128 bytes backwards; positive values up to 127 bytes forwards. (SJMP with 00H in the machine code offset byte will proceed with the following instruction).

In keeping with the 8051 assembly language goal of minimizing the number of instruction mnemonics, there is a "generic" form of the three jump instructions. ASM51 recognizes the mnemonic JMP as a "pseudo-instruction," translating it into the machine instructions LJMP, AJMP, or SJMP, depending on the destination address.

Like SJMP, all conditional jump instructions use relative addressing. JZ (Jump if Zero) and JNZ (Jump if Not Zero) monitor the state of the accumulator as implied by their names, while JC (Jump on Carry) and JNC (Jump on No Carry) test whether or not the carry flag is set. All four are two-byte instructions, with the same format as Figure 15.c. JB (Jump on Bit), JNB (Jump on No Bit) and JBC (Jump on Bit then Clear Bit) can test any status bit or input pin with a three byte instruction; the second byte specifies which bit to test and the third gives the relative offset value.

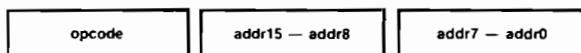
There are two subroutine-call instructions, LCALL (Long Call) and ACALL (Absolute Call). Each increments the P.C. to the first byte of the following instruction, then pushes it onto the stack (low byte first). Saving both bytes increments the stack pointer by two. The subroutine's starting address is encoded in the same ways as LJMP and AJMP. The generic form of the call operation is the mnemonic CALL, which ASM51 will translate into LCALL or ACALL as appropriate.

The return instruction RET pops the high- and low-order bytes of the program counter successively from the stack, decrementing the stack pointer by two. Program execution continues at the address previously pushed: the first byte of the instruction immediately following the call.

When an interrupt request is recognized by the 8051 hardware, two things happen. Program control is automatically "vectored" to one of the interrupt service routine starting addresses by, in effect, forcing the CPU to process an LCALL instead of the next instruction. This automatically stores the return address on the stack. (Unlike the 8048, no status information is automatically saved.)

Secondly, the interrupt logic is disabled from accepting any other interrupts from the same or lower priority. After completing the interrupt service routine, executing an RETI (Return from Interrupt) instruction will return execution to the point where the background program was interrupted — just like RET — while restoring the interrupt logic to its previous state.

a.) Long Jump (LJMP addr16):



b.) Absolute Jump (AJMP addr11):



c.) Short Jump (SJMP rel):



**Figure 15. Jump Instruction Machine Code Formats**

### Operate-and-branch Instructions — CJNE, DJNZ

Two groups of instructions combine a byte operation with a conditional jump based on the results.

CJNE (Compare and Jump if Not Equal) compares two byte operands and executes a jump if they disagree. The carry flag is set following the rules for subtraction: if the unsigned integer value of the first operand is less than that of the second it is set; otherwise, it is cleared. However, neither operand is modified.

The CJNE instruction provides, in effect, a one-instruction "case" statement. This instruction may be executed repeatedly, comparing the code variable to a list of "special case" value: the code segment following the instruction (up to the destination label) will be executed only if the operands match. Comparing the accumulator or a register to a series of constants is a convenient way to check for special handling or error conditions; if none of the cases match the program will continue with "normal" processing.

A typical example might be a word processing device which receives ASCII characters through the serial port and drives a thermal hard-copy printer. A standard routine translates "printing" characters to bit patterns, but control characters (<DEL>, <CR>, <LF>, <BEL>, <ESC>, or <SP>) must invoke corresponding special routines. Any other character with an ASCII code less than 20H should be translated into the <NUL> value, 00H, and processed with the printing characters.

#### Example 16—Case Statements Using CJNE

```

CHAR EQU R7 .CHARACTER CODE VARIABLE
;
INTP: CJNE CHAR, #7FH, INTP_1 ;(SPECIAL ROUTINE FOR RUBOUT CODE)
;           RET
INTP_1: CJNE CHAR, #07H, INTP_2 ;(SPECIAL ROUTINE FOR BELL CODE)
;           RET
INTP_2: CJNE CHAR, #0AH, INTP_3 ;(SPECIAL ROUTINE FOR LFEED CODE)
;           RET
INTP_3: CJNE CHAR, #0DH, INTP_4 ;(SPECIAL ROUTINE FOR RETURN CODE)
;           RET
INTP_4: CJNE CHAR, #1BH, INTP_5 ;(SPECIAL ROUTINE FOR ESCAPE CODE)
;           RET
INTP_5: CJNE CHAR, #20H, INTP_6 ;(SPECIAL ROUTINE FOR SPACE CODE)
;           RET
INTP_6: JC PRINTC ;JUMP IF CODE > 20H
        MOV #0, R0 ;REPLACE CONTROL CHARACTERS WITH
;           NULL CODE
        PRINTC ;PROCESS STANDARD PRINTING
;           CHARACTER
        RET

```

DJNZ (Decrement and Jump if Not Zero) decrements the register or direct address indicated and jumps if the result is not zero, without affecting any flags. This provides a simple means for executing a program loop a given number of times, or for adding a moderate time delay (from 2 to 512 machine cycles) with a single instruction. For example, a 99-usec. software delay loop can be added to code forcing an I/O pin low with only two instructions.

#### Example 17—Inserting a Software Delay with DJNZ

```

CLR    WR
MOV    R2, #49
DJNZ   R2, *
SETB   WR

```

The dollar sign in this example is a special character meaning "the address of this instruction." It is useful in eliminating instruction labels on the same or adjacent source lines. CJNE and DJNZ (like all conditional jumps) use program-counter relative addressing for the destination address.

### Stack Operations — PUSH, POP

The PUSH instruction increments the stack pointer by one, then transfers the contents of the single byte variable indicated (direct addressing only) into the internal RAM location addressed by the stack pointer. Conversely, POP copies the contents of the internal RAM location addressed by the stack pointer to the byte variable indicated, then decrements the stack pointer by one.

(Stack Addressing follows the same rules, and addresses the same locations as Register-indirect. Future microcomputers based on the MCS-51™ CPU could have up to 256 bytes of RAM for the stack.)

Interrupt service routines must not change any variable or hardware registers modified by the main program, or else the program may not resume correctly. (Such a change might look like a spontaneous random error.) Resources used or altered by the service routine (Accumulator, PSW, etc.) must be saved and restored to their previous value before returning from the service routine. PUSH and POP provide an efficient and convenient way to save register states on the stack.

#### Example 18—Use of the Stack for Status Saving on Interrupts

```

LOC_TMP EQU $ .REMEMBER LOCATION COUNTER
;
        ORG 0003H ;STARTING ADDRESS FOR INTERRUPT ROUTINE
        LJMP SERVER ;JUMP TO ACTUAL SERVICE ROUTINE LOCATED
;           ELSEWHERE
;
        ORG LOC_TMP .RESTORE LOCATION COUNTER
SERVER: PUSH PSW ;SAVE ACCUMULATOR (NOTE DIRECT ADDRESSING
;           NOTATION)
        PUSH B ;SAVE B REGISTER
        PUSH DPL ;SAVE DATA POINTER
        PUSH DPH
        MOV PSW, #00001000B ;SELECT REGISTER BANK 1
;
        POP DPH ;RESTORE REGISTERS IN REVERSE ORDER
        POP DPL
        POP B
        POP ACC
        POP PSW ;RESTORE PSW AND RE-SELECT ORIGINAL
;           REGISTER BANK
        RETI ;RETURN TO MAIN PROGRAM AND RESTORE
;           INTERRUPT LOGIC

```

If the SP register held 1FH when the interrupt was detected, then while the service routine was in progress the stack would hold the registers shown in Figure 16; SP would contain 26H.

The example shows the most general situation; if the service routine doesn't alter the B-register and data pointer, for example, the instructions saving and restoring those registers would not be necessary.

The stack may also pass parameters to and from subroutines. The subroutine can indirectly address the parameters derived from the contents of the stack pointer.

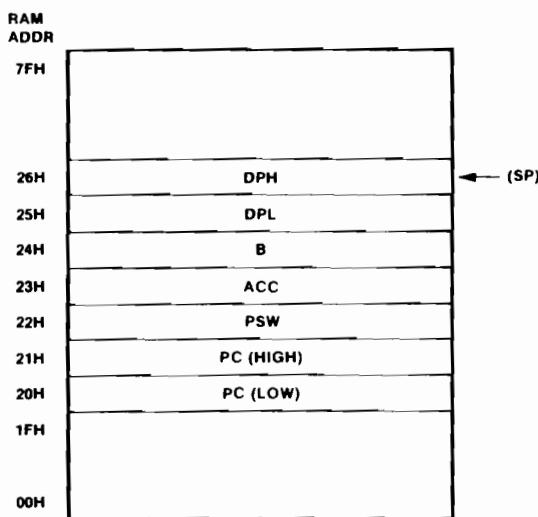


Figure 16. Stack contents during interrupt

One advantage here is simplicity. Variables need not be allocated for specific parameters, a potentially large number of parameters may be passed, and different calling programs may use different techniques for determining or handling the variables.

For example, the following subroutine reads out a parameter stored on the stack by the calling program, uses the low order bits to access a local look-up table holding bit patterns for driving the coils of a four phase stepper motor, and stores the appropriate bit pattern back in the same position on the stack before returning. The accumulator contents are left unchanged.

**Example 19—Passing Variable Parameters to Subroutines Using the Stack**

```

NXTPOS  MOV  RO, SP
DEC    RO  ; ACCESS LOCATION PARAMETER PUSHED INTO
DEC    RO  ; ACCUMULATOR
XCH   A, RO  ; READ INPUT PARAMETER AND SAVE
       ; ACCUMULATOR
ANL    A, #03H ; MASK ALL BUT LOW-ORDER TWO BITS
ADD    A, #2  ; ALLOW FOR OFFSET FROM MOVC TO TABLE
MOVC   A, @A+PC ; READ LOOK-UP TABLE ENTRY
XCH   A, RO  ; PASS BACK TRANSLATED VALUE AND RESTORE
       ; ACC
RET
STPTBL DB   01101111B ; POSITION 0
DB   01011111B ; POSITION 1
DB   10011111B ; POSITION 2
DB   10101111B ; POSITION 3

```

The background program may reach this subroutine with several different calling sequences, all of which PUSH a value before calling the routine and POP the result after. A motor on Port 1 may be initialized by placing the desired position (zero) on the stack before calling the subroutine and outputting the results directly to a port afterwards.

**Example 20—Sending and Receiving Data Parameters Via the Stack**

```

CLR    A
PUSH   ACC
CALL   NXTPOS
POP    P1

```

If the position of the motor is determined by the contents of variable POSM1 (a byte in internal RAM) and the position of a second motor on Port 2 is determined by the data input to the low-order nibble of Port 2, a six-instruction sequence could update them both.

**Example 21—Loading and Unloading Stack Direct from I/O Ports**

```

POSM1  EQU  $1
       PUSH  POSM1
       CALL  NXTPOS
       POP   P1
       PUSH  P2
       CALL  NXTPOS
       POP   P2

```

**Data Pointer and Table Look-up instructions —  
MOV, INC, MOVC, JMP**

The data pointer can be loaded with a 16-bit value using the instruction MOV DPTR, #data16. The data used is stored in the second and third instruction bytes, high-order byte first. The data pointer is incremented by INC DPTR. A 16-bit increment is performed; an overflow from the low byte will carry into the high-order byte. Neither instruction affects any flags.

The MOVC (Move Constant) instructions (MOVC A,@A+DPTR and MOVC A,@A+PC) read into the accumulator bytes of data from the program memory logical address space. Both use a form of indexed addressing: the former adds the unsigned eight-bit accumulator contents with the sixteen-bit data pointer register, and uses the resulting sum as the address from which the byte is fetched. A sixteen-bit addition is performed; a carry-out from the low-order eight bits may propagate through higher-order bits, but the contents of the DPTR are not altered. The latter form uses the incremented program counter as the "base" value instead of the DPTR (figure 17). Again, neither version affects the flags.

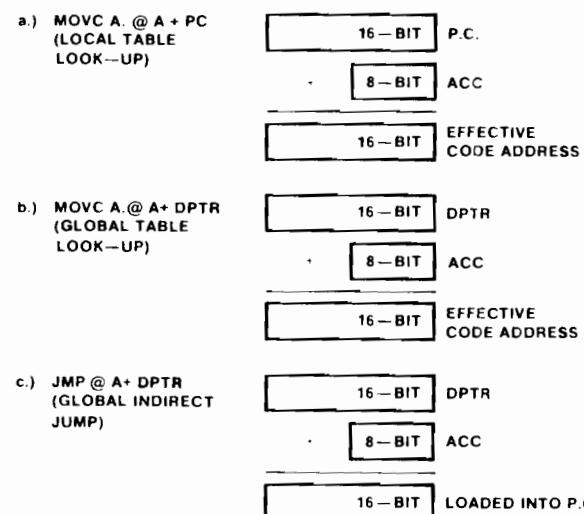


Figure 17. Operation of MOVC instructions

Each can be part of a three step sequence to access look-up tables in ROM. To use the DPTR-relative version, load the Data Pointer with the starting address of a look-up table; load the accumulator with (or compute) the index of the entry desired; and execute MOVC A,@A+DPTR. Unlike the similar MOVP3 instructions in the 8048, the table may be located anywhere in program memory. The data pointer may be loaded with a constant for short tables. Or to allow more complicated data structures, or tables with more than 256 entries, the values for DPH and DPL may be computed or modified with the standard arithmetic instruction set.

The PC-relative version has the advantage of not affecting the data pointer. Again, a look-up sequence takes three steps: load the accumulator with the index; compensate for the offset from the look-up instruction to the start of the table by adding the number of bytes separating them to the accumulator; then execute the MOVC A,@A+PC instruction.

Let's look at a non-trivial situation where this instruction would be used. Some applications store large multi-dimensional look-up tables of dot matrix patterns, non-linear calibration parameters, and so on in a linear (one-dimensional) vector in program memory. To retrieve data from the tables, variables representing matrix indices must be converted to the desired entry's memory address. For a matrix of dimensions (MDIMEN x NDIMEN) starting at address BASE and respective indices INDEXI and INDEXJ, the address of element (INDEXI, INDEXJ) is determined by the formula,

$$\text{Entry Address} = \text{BASE} + (\text{NDIMEN} \times \text{INDEXI}) + \text{INDEXJ}$$

The code shown below can access any array with less than 255 entries (i.e., an 11x21 array with 231 elements). The table entries are defined using the Data Byte ("DB") directive, and will be contained in the assembly object code as part of the accessing subroutine itself.

#### Example 22—Use of MPY and Data Pointer Instructions to Access Entries from a Multi-dimensional Look-Up Table in ROM

```

;MATRIX1 LOAD CONSTANT READ FROM TWO DIMENSIONAL LOOK-UP
; TABLE IN PROGRAM MEMORY INTO ACCUMULATOR
; USING LOCAL TABLE LOOK-UP INSTRUCTION. 'MOVC' A, EA+PC
; THE TOTAL NUMBER OF TABLE ENTRIES IS ASSUMED TO
; BE SMALL, I.E. LESS THAN ABOUT 250 ENTRIES
; TABLE USED IN THIS EXAMPLE IS ( 11 X 21 )
; DESIRED ENTRY ADDRESS IS GIVEN BY THE FORMULA,
; [ (BASE ADDRESS) + (21 X INDEXI) + (INDEXJ) ]

INDEXI EQU R6 ; FIRST COORDINATE OF ENTRY (0-10).
INDEXJ EQU 23H ; SECOND COORDINATE OF ENTRY (0-20)

;MATRIX1 MOV A, INDEXI
; MOV B, #21
; MUL AB
; ADD A, INDEXJ
; ALLOW FOR INSTRUCTION BYTE BETWEEN "MOVC" AND
; ENTRY (0,0)
; INC A
; MOVC A, @A+PC
; RET

BASE1 DB 1 ;(entry 0,0)
DB 2 ;(entry 0,1)
DB 21 ;(entry 0,20)
DB 22 ;(entry 1,0)
DB 42 ;(entry 1,20)
DB 231 ;(entry 10,20)

```

There are several different means for branching to sections of code determined or selected at run time. (The single destination addresses incorporated into conditional and unconditional jumps are, of course, determined at assembly time). Each has advantages for different applications.

The most common is an N-way conditional jump based on some variable, with all of the potential destinations known at assembly time. One of a number of small routines is selected according to the value of an index variable determined while the program is running. The most efficient way to solve this problem is with the MOVC and an indirect jump instruction, using a short table of one byte offset values in ROM to indicate the relative starting addresses of the several routines.

JMP @A+DPTR is an instruction which performs an indirect jump to an address determined during program execution. The instruction adds the eight-bit unsigned accumulator contents with the contents of the sixteen-bit data pointer, just like MOVC A,@A+DPTR. The resulting sum is loaded into the program counter and is used as the address for subsequent instruction fetches. Again, a sixteen-bit addition is performed; a carry out from the low-order eight bits may propagate through the higher-order bits. In this case, neither the accumulator contents nor the data pointer is altered.

The example subroutine below reads a byte of RAM into the accumulator from one of four alternate address spaces, as selected by the contents of the variable MEMSEL. The address of the byte to be read is determined by the contents of R0 (and optionally R1). It might find use in a printing terminal application, where four different model printers all use the same ROM code but use different types and sizes of buffer memory for different speeds and options.

#### Example 23—N-Way Branch and Computed Jump Instructions via JMP @ ADPTR

```

MEMSEL EQU R3
;
JUMP_4: MOV A, MEMSEL
        MOV DPTR, @JMPtbl
        MOVC A, @A+DPTR
        JMP @A+DPTR
JMPtbl: DB MEMSP0-JMPtbl
        DB MEMSP1-JMPtbl
        DB MEMSP2-JMPtbl
        DB MEMSP3-JMPtbl
MEMSP0: MOV A, R0 ;READ FROM INTERNAL RAM
        RET
MEMSP1: MOVX A, R0 ;READ FROM 256 BYTES OF EXTERNAL RAM
        RET
MEMSP2: MOV DPL, R0
        MOV DPH, R1
        MOVX A, @DPTR ;READ FROM 64K BYTES OF EXTERNAL RAM
        RET
MEMSP3: MOV A, R1
        ANL A, #07H
        ANL P1, #1111000B
        DRL P1, A
        MOVX A, R0 ;READ FROM 4K BYTES OF EXTERNAL RAM
        RET

```

Note that this approach is suitable whenever the size of jump table plus the length of the alternate routines is less than 256 bytes. The jump table and routines may be located anywhere in program memory, independent of 256-byte program memory pages.

For applications where up to 128 destinations must be selected, all of which reside in the same 2K page of program memory which may be reached by the two-byte absolute jump instructions, the following technique may be used. In the above mentioned printing terminal example, this sequence could "parse" 128 different codes for ASCII characters arriving via the 8051 serial port.

**Example 24—N-Way Branch with 128 Optional Destinations**

```

OPTION EQU R3
;
JMP128 MOV A, OPTION
RL A, 2             ; MULTIPLY BY 2 FOR 2 BYTE JUMP TABLE
MOV DPTR, #INSTBL ; FIRST ENTRY IN JUMP TABLE
JMP @A+DPTR        ; JUMP INTO JUMP TABLE
;
INSTBL: AJMP PROC00 ; 128 CONSECUTIVE
AJMP PROC01 ; AJMP INSTRUCTIONS
AJMP PROC02
;
AJMP PROC7E
AJMP PROC7F
;
```

The destinations in the jump table (PROC00-PROC7F) are not all necessarily unique routines. A large number of special control codes could each be processed with their own unique routine, with the remaining printing characters all causing a branch to a common routine for entering the character into the output queue.

In those rare situations where even 128 options are insufficient, or where the destination routines may cross a 2K page boundary, the above approach may be modified slightly as shown below.

**Example 25—256-Way Branch Using Address Look-Up Tables**

```

RTMP EQU R7
;
JMP256 MOV DPTR, #ADRTBL ; FIRST ENTRY IN TABLE OF ADDRESSES
MOV A, OPTION
CLR C
RLC A, 2             ; MULTIPLY BY 2 FOR 2 BYTE JUMP TABLE
JNC LOW128
INC DPH
;
LOW128 MOV RTMP, A ; SAVE ACC FOR HIGH BYTE READ
MOV A, @A+DPTR ; READ LOW BYTE FROM JUMP TABLE
XCH, A, RTMP
INC A
MOVC A, @A+DPTR ; GET LOW-ORDER BYTE FROM TABLE
PUSH ACC
MOV A, RTMP
MOVC A, @A+DPTR ; GET HIGH-ORDER BYTE FROM TABLE
PUSH ACC
;
THE TWO ACC PUSHES HAVE PRODUCED
A "RETURN ADDRESS" ON THE STACK WHICH CORRESPONDS
TO THE DESIRED STARTING ADDRESS
IT MAY BE REACHED BY POPPING THE STACK
INTO THE PC
RET
;
ADRTBL: DW PROC00 ; UP TO 256 CONSECUTIVE DATA
DW PROC01 ; WORDS INDICATING STARTING ADDRESSES
DW PROCFF
;
DUMMY CODE ADDRESS DEFINITIONS NEEDED BY ABOVE
TWO EXAMPLES
;
PROC00 NOP
PROC01 NOP
PROC02 NOP
PROC7E NOP
PROC7F NOP
PROCFF NOP
;
```

**4. BOOLEAN PROCESSING INSTRUCTIONS**

The commonly accepted terms for tasks at either end of the computational vs. control application spectrum are, respectively, "number-crunching" and "bit-banging".

Prior to the introduction of the MCS-51™ family, nice number-crunchers made bad bit-bangers and vice versa. The 8051 is the industry's first single-chip microcomputer designed to crunch **and** bang. (In some circles, the latter technique is also referred to as "bit-twiddling". Either is correct.)

**Direct Bit Addressing**

A number of instructions operate on Boolean (one-bit) variables, using a direct bit addressing mode comparable to direct byte addressing. An additional byte appended to the opcode specifies the Boolean variable, I/O pin, or control bit used. The state of any of these bits may be tested for "true" or "false" with the conditional branch instructions JB (Jump on Bit) and JNB (Jump on Not Bit). The JBC (Jump on Bit and Clear) instruction combines a test-for-true with an unconditional clear.

As in direct byte addressing, bit 7 of the address byte switches between two physical address spaces. Values between 0 and 127 (00H-7FH) define bits in internal RAM locations 20H to 2FH (Figure 18a); address bytes between 128 and 255 (80H-0FFH) define bits in the 2 x "special-function" register address space (Figure 18b). If no 2 x "special-function" register corresponds to the direct bit address used the result of the instruction is undefined.

Bits so addressed have many wondrous properties. They may be set, cleared, or complemented with the two byte instructions SETB, CLR, or CPL. Bits may be moved to and from the carry flag with MOV. The logical ANL and ORL functions may be performed between the carry and either the addressed bit or its complement.

**Bit Manipulation Instructions — MOV**

The "MOV" mnemonic can be used to load an addressable bit into the carry flag ("MOV C, bit") or to copy the state of the carry to such a bit ("MOV bit, C"). These instructions are often used for implementing serial I/O algorithms via software or to adapt the standard I/O port structure.

It is sometimes desirable to "re-arrange" the order of I/O pins because of considerations in laying out printed circuit boards. When interfacing the 8051 to an immediately adjacent device with "weighted" input pins, such as keyboard column decoder, the corresponding pins are likely to be not aligned (Figure 19).

There is a trade-off in "scrambling" the interconnections with either interwoven circuit board traces or through software. This is extremely cumbersome (if not impossible) to do with byte-oriented computer architectures. The 8051's unique set of Boolean instructions makes it simple to move individual bits between arbitrary locations.

a.) RAM Bit Addresses.

RAM BYTE	(MSB)								(LSB)							
7FH																
2FH	7F	7E	7D	7C	7B	7A	79	78								
2EH	77	76	75	74	73	72	71	70								
2DH	6F	6E	6D	6C	6B	6A	69	68								
2CH	67	66	65	64	63	62	61	60								
2BH	5F	5E	5D	5C	5B	5A	59	58								
2AH	57	56	55	54	53	52	51	50								
29H	4F	4E	4D	4C	4B	4A	49	48								
28H	47	46	45	44	43	42	41	40								
27H	3F	3E	3D	3C	3B	3A	39	38								
26H	37	36	35	34	33	32	31	30								
25H	2F	2E	2D	2C	2B	2A	29	28								
24H	27	26	25	24	23	22	21	20								
23H	1F	1E	1D	1C	1B	1A	19	18								
22H	17	16	15	14	13	12	11	10								
21H	0F	0E	0D	0C	0B	0A	09	08								
20H	07	06	05	04	03	02	01	00								
1FH	Bank 3															
18H																
17H	Bank 2															
10H																
0FH	Bank 1															
08H																
07H	Bank 0															
00H																

b.) Hardware Register Bit Addresses.

Direct Byte Address	(MSB)								Bit Addresses								Hardware Register Symbol
0FFH																	
0FOH	F7 F6 F5 F4 F3 F2 F1 F0																B
0E0H	E7 E6 E5 E4 E3 E2 E1 E0																ACC
0D0H	D7 D6 D5 D4 D3 D2 D1 D0																PSW
0B8H	— — — BC BB BA B9 B8																IP
0B0H	B7 B6 B5 B4 B3 B2 B1 B0																P3
0A8H	AF — — AC AB AA A9 A8																IE
0A0H	A7 A6 A5 A4 A3 A2 A1 A0																P2
98H	9F 9E 9D 9C 9B 9A 99 98																SCON
90H	97 96 95 94 93 92 91 90																P1
88H	8F 8E 8D 8C 8B 8A 89 88																TCON
80H	87 86 85 84 83 82 81 80																P0

Figure 18. Bit Address Maps

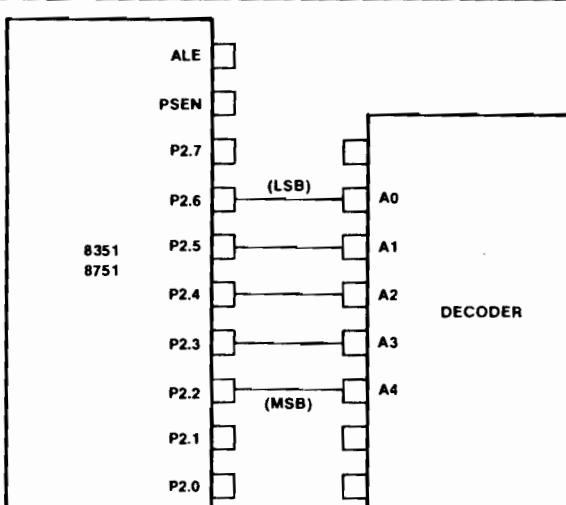


Figure 19. "Mismatch" Between I/O port and Decoder

## Example 26—Re-ordering I/O Port Configuration

```

OUT_PZ: RRC    A    , MOVE ORIGINAL ACC 0 INTO CY
        MOV    P2.6,C , STORE CARRY TO PIN P26
        RRC    A    , MOVE ORIGINAL ACC 1 INTO CY
        MOV    P2.5,C , STORE CARRY TO PIN P25
        RRC    A    , MOVE ORIGINAL ACC 2 INTO CY
        MOV    P2.4,C , STORE CARRY TO PIN P24
        RRC    A    , MOVE ORIGINAL ACC 3 INTO CY
        MOV    P2.3,C , STORE CARRY TO PIN P23
        RRC    A    , MOVE ORIGINAL ACC 4 INTO CY
        MOV    P2.2,C , STORE CARRY TO PIN P22
        RET
  
```

## Solving Combinatorial Logic Equations — ANL, ORL

Virtually all hardware designers are familiar with the problem of solving complex functions using combinatorial logic. The technologies involved may vary greatly, from multiple contact relay logic, vacuum tubes, TTL, or CMOS to more esoteric approaches like fluidics, but in each case the goal is the same: a Boolean (true/false) function is computed on a number of Boolean variables.

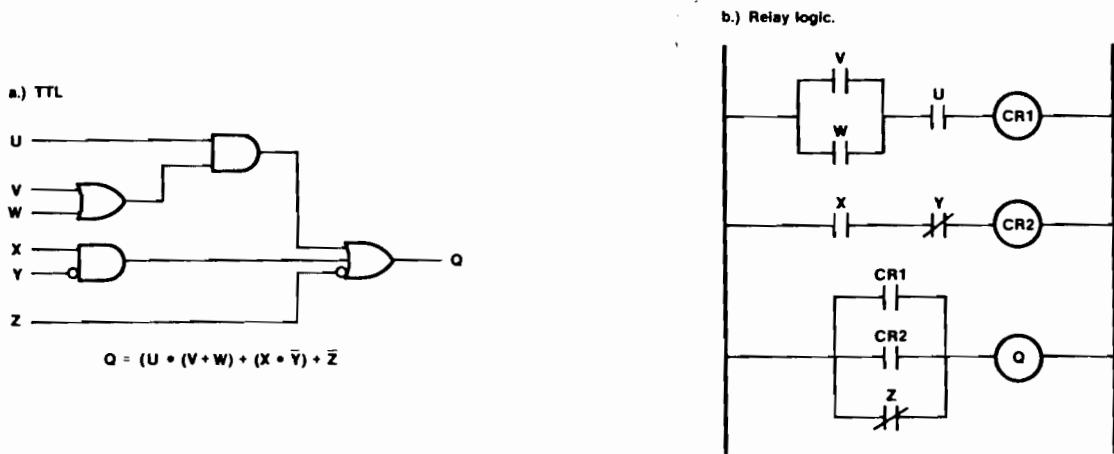


Figure 20. Implementations of Boolean functions

Figure 20 shows the logic diagram for an arbitrary function of six variables named U through Z using standard logic and relay logic symbols. Each is a solution of the equation,

$$Q = (U * (V + W)) + (X * Y) + Z$$

(While this equation could be reduced using Karnaugh Maps or algebraic techniques, that is not the purpose of this example. Even a minor change to the function equation would require re-reducing from scratch.)

Most digital computers can solve equations of this type with standard word-wide logical instructions and conditional jumps. Still, such software solutions seem somewhat sloppy because of the many paths through the program the computation can take.

Assume U and V are input pins being read by different input ports, W and X are status bits for two peripheral controllers (read as I/O ports), and Y and Z are software flags set or cleared earlier in the program. The end result must be written to an output pin on some third port.

For the sake of comparison we will implement this function with software drawn from three proper subsets of the MCS-51™ instruction set. The first two implementations follow the flow chart shown in Figure 21. Program flow would embark on a route down a test-and-branch tree and leaves either the "True" or "Not True" exit ASAP. These exits then write the output port with the data previously written to the same port with the result bit respectively one or zero.

In the first case, we assume there are no instructions for addressing individual bits other than special flags like the carry. This is typical of many older microprocessors and mainframe computers designed for number-crunching. MCS-51™ mnemonics are used here, though for most other machines the issue would be even further clouded by their use of operation-specific mnemonics like

INPUT, OUTPUT, LOAD, STORE, etc.. instead of the universal MOV.

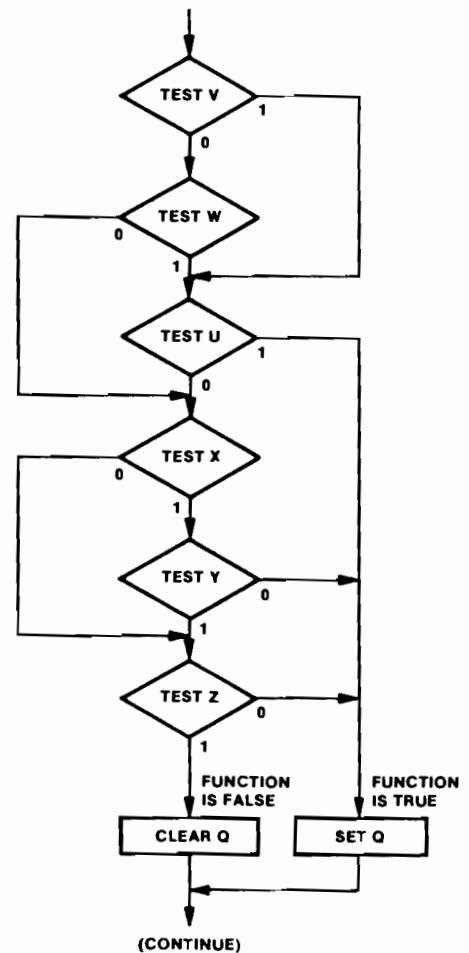


Figure 21. Flow chart for tree-branching logic implementation

**Example 27—Software Solution to Logic Function of Figure 20, Using only Byte-Wide Logical Instructions**

```
; BFUNC1 SOLVE A RANDOM LOGIC FUNCTION OF 6
; VARIABLES BY LOADING AND MASKING THE APPROPRIATE
; BITS IN THE ACCUMULATOR, THEN EXECUTING CONDITIONAL
; JUMPS BASED ON ZERO CONDITION
; (APPROACH USED BY BYTE-ORIENTED ARCHITECTURES)
; BYTE AND MASK VALUES CORRESPOND TO RESPECTIVE
; BYTE ADDRESS AND BIT POSITION
;
OUTBUF EQU 22H .OUTPUT PIN STATE MAP
;
TESTV: MOV A, P2
ANL A, #00000100B
JNZ TESTU
MOV A, TCON
ANL A, #00100000B
JZ TESTX
;
TESTU: MOV A, P1
ANL A, #00000010B
JNZ SETG
TESTX: MOV A, TCON
ANL A, #00000100B
JZ TESTZ
MOV A, 20H
ANL A, #000000001B
JZ SETG
TESTZ: MOV A, 21H
ANL A, #00000010B
JZ SETG
CLRQ: MOV A, OUTBUF
ANL A, #11110111B
JMP OUTQ
SETQ: MOV A, OUTBUF
ORL A, #00001000B
OUTQ: MOV OUTBUF, A
MOV P3, A
```

Cumbersome, to say the least, and error prone. It would be hard to prove the above example worked in all cases without an exhaustive test.

Each move/mask/conditional jump instruction sequence may be replaced by a single bit-test instruction thanks to direct bit addressing. But the algorithm would be equally convoluted.

**Example 28—Software Solution to Logic Function of Figure 20, Using only Bit-Test Instructions**

```
; BFUNC2 SOLVE A RANDOM LOGIC FUNCTION OF 6
; VARIABLES BY DIRECTLY POLLING EACH BIT.
; (APPROACH USING MCS-51 UNIQUE BIT-TEST
; INSTRUCTION CAPABILITY)
; SYMBOLS USED IN LOGIC DIAGRAM ASSIGNED TO
; CORRESPONDING 8051 BIT ADDRESSES
;
U BIT P1.1
V BIT P2.2
W BIT TFO
X BIT IE1
Y BIT 20H.0
Z BIT 21H.1
G BIT P3.3
;
TEST_V: JB V, TEST_U
JNB W, TEST_X
TEST_U: JB W, SET_G
TEST_X: JNB X, TEST_Z
TEST_Z: JNB Y, SET_G
CLR_G: CLR G
JMP NXTTST
SET_G: SETB G
NXTTST: (CONTINUATION OF PROGRAM)
```

A more elegant and efficient 8051 implementation uses the Boolean ANL and ORL functions to generate the output function using straight-line code. These instructions perform the corresponding logical operations between the carry flag ("Boolean Accumulator") and the addressed bit, leaving the result in the carry. Alternate forms of each instruction (specified in the assembly language by placing a slash before the bit name) use the complement of the bit's state as the input operand.

These instructions may be "strung together" to simulate a multiple input logic gate. When finished, the carry flag contains the result, which may be moved directly to the destination or output pin. No flow chart is needed — it is simple to code directly from the logic diagrams in Figure 20.

**Example 29—Software Solution to Logic Function of Figure 20, Using the MCS-51 (TM) Unique Logical Instructions on Boolean Variables**

```
; BFUNC3 SOLVE A RANDOM LOGIC FUNCTION OF 6
; VARIABLES USING STRAIGHT-LINE LOGICAL INSTRUCTIONS
; ON MCS-51 BOOLEAN VARIABLES
;
MOV C, V
ORL C, W .OUTPUT OF OR GATE
ANL C, U .OUTPUT OF TOP AND GATE
MOV F0, C .SAVE INTERMEDIATE STATE
MOV C, X
ANL C, /Y .OUTPUT OF BOTTOM AND GATE
ORL C, F0 .INCLUDE VALUE SAVED ABOVE
ORL C, /Z .INCLUDE LAST INPUT VARIABLE
MOV G, C .OUTPUT COMPUTED RESULT
```

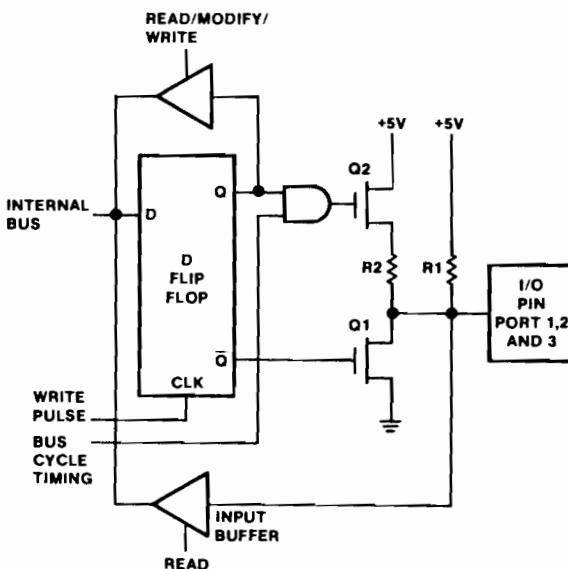
Simplicity itself. Fast, flexible, reliable, easy to design, and easy to debug.

The Boolean features are useful and unique enough to warrant a complete Application Note of their own. Additional uses and ideas are presented in Application Note AP-70, **Using the Intel® MCS-51® Boolean Processing Capabilities**, publication number 121519.

## 5. ON-CHIP PERIPHERAL FUNCTION OPERATION AND INTERFACING

### I/O Ports

The I/O port versatility results from the "quasi-bidirectional" output structure depicted in Figure 22. (This is effectively the structure of ports 1, 2, and 3 for normal I/O operations. On port 0 resistor R2 is disabled except during multiplexed bus operations, providing



**Figure 22. Pseudo-bidirectional I/O port circuitry**

AFN-01502A-30

essentially open-collector outputs. For full electrical characteristics see the User's Manual.)

An output latch bit associated with each pin is updated by direct addressing instructions when that port is the destination. The latch state is buffered to the outside world by R1 and Q1, which may drive a standard TTL input. (In TTL terms, Q1 and R1 resemble an open-collector output with a pull-up resistor to Vcc.)

R2 and Q2 represent an "active pull-up" device enabled momentarily when a 0 previously output changes to a 1. This "jerks" the output pin to a 1 level more quickly than the passive pull-up, improving rise-time significantly if the pin is driving a capacitive load. Note that the active pull-up is only activated on 0-to-1 transitions at the output latch (unlike the 8048, in which Q2 is activated whenever a 1 is written out).

Operations using an input port or pin as the source operand use the logic level of the pin itself, rather than the output latch contents. This level is affected by both the microcomputer itself and whatever device the pin is connected to externally. The value read is essentially the "OR-tied" function of Q1 and the external device. If the external device is high-impedance, such as a logic gate input or a three state output in the third state, then reading a pin will reflect the logic level previously output. To use a pin for input, the corresponding output latch must be set. The external device may then drive the pin with either a high or low logic signal. Thus the same port may be used as both input and output by writing ones to all pins used as inputs on output operations, and ignoring all pins used as output on an input operation.

In one operand instructions (INC, DEC, DJNZ and the Boolean CPL) the output latch rather than the input pin level is used as the source data. Similarly, two operand instructions using the port as both one source and the destination (ANL, ORL, XRL) use the output latches. This ensures that latch bits corresponding to pins used as inputs will not be cleared in the process of executing these instructions.

The Boolean operation JBC tests the output latch bit, rather than the input pin, in deciding whether or not to jump. Like the byte-wise logical operations, Boolean operations which modify individual pins of a port leave the other bits of the output latch unchanged.

A good example of how these modes may play together may be taken from the host-processor interface expected by an 8243 I/O expander. Even though the 8051 does not include 8048-type instructions for interfacing with an 8243, the parts can be interconnected (Figure 23) and the protocol may be emulated with simple software.

### Example 30—Mixing Parallel Output, Input Control Strobes on Port 2

```

;IN8243 INPUT DATA FROM AN 8243 I/O EXPANDER
;CONNECTED TO P23-P20
;P25 & P24 MIMIC CS/ & PROG
;P27-P26 USED AS INPUTS
;PORT TO BE READ IN ACC

IN8243 ORL    A,#11010000B
MOV    P2,A    ;OUTPUT INSTRUCTION CODE
CLR    P2.4    ;FALLING EDGE OF PROG
ORL    P2.000001111B ;SET FOR INPUT
MOV    A,P2    ;READ INPUT DATA
SETB  P2.4    ;RETURN PROG HIGH
SETB  P2.5    ;DE-SELECT CHIP

```

### Serial Port and Timer applications

Configuring the 8051's Serial Port for a given data and protocol requires essentially three short sections of software. On power-up or hardware reset the serial port and timer control words must be initialized to appropriate values. Additional software is also needed in the transmit routine to load the serial port data and in the receive routine to unload the data as it is received.

This is best illustrated through an arbitrary example. Assume the 8051 will communicate with a serial device operating at 2400 baud (bits per second). Each character transmitted is transmitted as seven data bits, odd parity, and one stop bit. This results in a character rate of 2400/11=216 characters per second.

For the sake of clarity, the transmit and receive subroutines are driven by simple-minded software polling code rather than interrupts. (It might help to go back to Figures 7-9 showing the control word for the serial port.) The serial port must be initialized to 8-bit UART mode (M0, M1=01), enabled to receive all messages (REN=1), and the flag indicating that the transmit register is free for more data will be artificially set in order for the output software to know the output register is available. This can all be set up with one instruction.

### Example 31—Serial Port Mode and Control

```

;SPINIT INITIALIZE SERIAL PORT
;FOR 8-BIT UART MODE
;& SET TRANSMIT READY FLAG
SPINIT: MOV    SCON, #01010010B

```

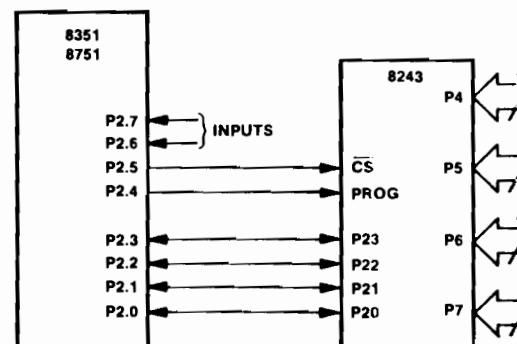


Figure 23. Connecting an 8051 with an 8243 I/O Expander

Timer 1 will be used in auto-reload mode as a data rate generator. To achieve a data rate of 2400 baud, the timer must divide the 1 MHz internal clock by 32 x (desired data rate):

$$\frac{1 \times 10^6}{(32)(2400)}$$

which equals 13.02 rounded down to 13 instruction cycles. The timer must reload the value -13, or 0F3H. (ASM51 will accept both the signed decimal or hexa-decimal representations.)

#### Example 32—Initializing Timer Mode and Control Bits

```
; T1INIT INITIALIZE TIMER 1 FOR
; AUTO-RELOAD AT 32*2400 HZ.
; (TO USED AS GATED 16-BIT COUNTER)

T1INIT: MOV TCON, #11010010B
        MOV TH1, #13
        SETB TR1
```

A simple subroutine to transmit the character passed to it the accumulator must first compute the parity bit, insert it into the data byte, wait until the transmitter is available, output the character, and return. This is nearly as easy said as done.

#### Example 33—Code for UART Output, Adding Parity, Transmitter Loading

```
; SP_OUT ADD ODD PARITY TO ACC AND
; TRANSMIT WHEN SERIAL PORT READY.

SP_OUT: MOV C, P
        CPL C
        MOV ACC, 7.C
        JNB TI, *
        CLR TI
        MOV SBUF, A
        RET
```

A simple minded routine to wait until a character is received, set the carry flag if there is an odd-parity error, and return the masked seven-bit code in the accumulator is equally short.

#### Example 34—Code for UART Reception and Parity Verification

```
; SP_IN INPUT NEXT CHARACTER FROM SERIAL PORT
; SET CARRY IFF ODD-PARITY ERROR

SP_IN: JNB RI, $
        CLR RI
        MOV A, SBUF
        MOV C, P
        CPL C
        ANL A, #7FH
        RET
```

## 6. SUMMARY

This Application Note has described the architecture, instruction set, and on-chip peripheral features of the first three members of the MCS-51™ microcomputer family. The examples used throughout were admittedly (and necessarily) very simple. Additional examples and techniques may be found in the MCS-51™ User's Manual and other application notes written for the MCS-48™ and MCS-51™ families.

Since its introduction in 1977, the MCS-48™ family has become the industry standard single-chip microcomputer. The MCS-51™ architecture expands the addressing capabilities and instruction set of its predecessor while ensuring flexibility for the future, and maintaining basic software compatibility with the past.

Designers already familiar with the 8048 or 8049 will be able to take with them the education and experience gained from past designs as ever-increasing system performance demands force them to move on to state-of-the-art products. Newcomers will find the power and regularity of the 8051 instruction set an advantage in streamlining both the learning and design processes.

Microcomputer system designers will appreciate the 8051 as basically a single-chip solution to many problems which previously required board-level computers. Designers of real-time control systems will find the high execution speed, on-chip peripherals, and interrupt capabilities vital in meeting the timing constraints of products previously requiring discrete logic designs. And designers of industrial controllers will be able to convert ladder diagrams directly from tested-and-true TTL or relay-logic designs to microcomputer software, thanks to the unique Boolean processing capabilities.

It has not been the intent of this note to gloss over the difficulty of designing microcomputer-based systems. To be sure, the hardware and software design aspects of any new computer system are nontrivial tasks. However, the system speed and level of integration of the MCS-51™ microcomputers, the power and flexibility of the instruction set, and the sophisticated assembler and other support products combine to give both the hardware and software designer as much of a head start on the problem as possible.



INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, CA 95051 (408) 987-8080